USENIX

UNIX SECURITY SYMPOSIUM IV PROCEEDINGS

Autumn
1993

U

US

USE

U NIX

USENIX

SYMPOSIUM PROCEEDINGS

UNIX Security IV

October 4-6, 1993
Santa Clara, California

For additional copies of these proceedings contact

USENIX Association
Ninth Street, Suite
Berkeley, CA 94710 U.S.A.

The price is $15 for members and $20 for nonmembers.

Outside the U.S.A and Canada,
please add $9 per copy for postage (via air printed matter).

Past USENIX Security Proceedings
(price: member/nonmember)

UNIX Security III	September 1992	Baltimore, MD	$30/39
UNIX Security II	August 1990	Portland, OR	$13/16
UNIX Security	August 1988	Portland, OR	$7/7

ISBN 1-880446-55-3

This volume is published as a collective work

USENIX acknowledges all trademarks appearing herein.

Printed on the United States of America on 50% recycled paper, 10-15% post-consumer waste.

UNIX Security Symposium IV
Proceedings

Sponsored by the USENIX Association in cooperation with the Computer Emergency Response Team (CERT) and ACM SIGSAC

October 4-6, 1993
Santa Clara, California, U.S.A.

4th UNIX SECURITY SYMPOSIUM
Santa Clara, California
October 4 - 6, 1993

Table of Contents

Program Committee

CryptoLib: Cryptography in Software

John B. Lacy [1]
Donald P. Mitchell [2]
William M. Schell [3]

AT&T Bell Laboratories

ABSTRACT

It is generally assumed that efficient implementations of public and private key cryptosystems such as RSA and DES are possible only in hardware or at least coded in very nonportable assembly language for particular processors. However hardware implementations are hardly ubiquitous and the overhead of assembly implementations is prohibitive. A portable and efficient software implementation is thus needed if we are to have the necessary set of tools with which to research privacy and security issues and to implement related protocols in the contexts for which they are intended.

As a result of upward trends in processor strength, it has become clear that for some kinds of applications, public key systems such as RSA and private key systems such as DES can indeed be implemented efficiently enough in portable software. Several questions arise: To what kinds of applications are we restricted? Should any part of the implementation be done in assembly language? What kinds of interfaces and hooks are necessary for protocol design and implementation?

CryptoLib is a portable and efficient library of routines necessary for public and private key systems. It written entirely in C. Assembly modules exist for some architectures for more efficient handling of 32bit primitive multiplications but the code does not depend on these modules.

What follows is a description of the library contents, some implementation details and some execution times for various architectures. Next is a discussion of applications which can withstand overhead imposed by the efficiency of such a library. Last is a brief discussion of our ongoing research into building a crypto-protocol layer and its interface to **CryptoLib**.

1. Introduction

With the capacity of communications channels increasing at the current rates and with the kinds of electronic services becoming more varied and multidimensional, there is also coming a greater tendency to store and forward sensitive, semi-private or very private information. The sorts of services which come to mind include cellular phones, multipurpose databases, interactive computer shopping services, electronic mail, contract negotiation, electronic funds transfer, digital cash, etc. With this tendency there is an increasing interest in protecting the privacy, integrity and security of communications channels and the privacy and possibly anonymity of the people involved in transactions.

Many protocols have been devised which provide varying degrees of privacy and security. These protocols make use of cryptosystems which may be divided roughly into two groups: private key

[1] lacy@research.att.com

[2] don@research.att.com

[3] bill@research.att.com

and public key. It is generally assumed that efficient implementations of these systems are possible only in hardware or at least coded in very nonportable assembly language for particular processors. However, hardware implementations are hardly ubiquitous and the overhead of assembly implementations is prohibitive. A portable and efficient software implementation is thus needed if we are to have the necessary set of tools with which to research privacy and security issues and to implement related protocols in the contexts for which they are intended.

As a result of upward trends in processor strength, it has become clear that for some kinds of applications, public key systems such as RSA and private key systems such as DES can indeed be implemented efficiently enough in portable software. Several questions arise: To what kinds of applications are we restricted? Should any part of the implementation be done in assembly language? What kinds of interfaces and hooks are necessary for protocol design and implementation?

CryptoLib is a portable and efficient library of routines necessary for public and private key systems. It written entirely in C. Assembly modules exist for some architectures for more efficient handling of 32bit primitive multiplications. The code does not depend on these modules. A list of the tools included in **CryptoLib** follows:

Big integer creation and manipulation tools.

Big Arithmetic functions: addition, subtraction,
division, multiplication, shifting.

Modular exponentiation and multiplication and the modulo operation.

Bitwise AND, OR and XOR (modulo 2 addition).

Chinese Remainder Theorem speedup of exponentiation.

Quadratic residue test, square root (mod prime),
square root (mod composite of two primes)

Random number generators -- pseudo and true.

Prime number generator and Primality test.

Euclid's Extended Greatest Common Divisor algorithm.

DES encryption and decryption tools.

RSA [4] key generation, encryption and decryption tools
and signature generation and verification tools.

El Gamal key generation, encryption and decryption tools
and signature generation and verification tools.

NIST key generation and Digital Signature Algorithm (DSA [5]).

NIST Secure Hash Standard (SHS)

[4] The RSA public-key encryption method is patented (Rivest, et. al. U.S. Patent 4,405,829, issued 9/20/83).

[5] The DSA is patented (U.S. Patent 5,231,668 , issued 7/27/93).

MD5 Message Digest tools (RSA, Inc.)

We now describe the library contents, some implementation details and some execution times for various architectures. Next is a discussion of applications which can withstand overhead imposed by the efficiency of such a library. Last is a brief discussion of our ongoing research into building a crypto-protocol layer and its interface to **CryptoLib**.

2. Implementation

2.1 Bigmath

2.1.1 **Big Number Representation and Basic Arithmetic:** Numbers of arbitrary length may be represented as

$$B = \sum_{i=0}^{i=n-1} b_i * r^i$$

For this library r is assumed to be 2^{32} so that, for example, the hexadecimal number $0xFFABCC12DDE3AB123456$ can be represented as

$$0xFFAB*r^2 + 0xCC12DDE3*r^1 + 0xAB123456*r^0.$$

Big numbers are represented in our library using the following structure:

```
struct bignum {
        Sign sign; /* POS or NEG */
        long length;
        unsigned long space;
        unsigned long *num;
} Bignum;
```

That is, *Bignums* have a sign, a length, a pointer to the bits which comprise the actual number and a space allocation counter. Manipulating big numbers reduces to the problem of manipulating large arrays. Efficiency is critical. We provide routines for creating and initializing bignums, for converting character arrays to bignums and for converting bignums to character arrays.

With the exception of multiplication and exponentiation, basic arithmetic functions are implemented in a straightforward O(n) manner. We provide addition, subtraction, division (which returns both the quotient and remainder), modulo, right and left shifting, bitwise AND, OR and XOR (modulo 2 addition), comparison and copy functions.

2.1.2 **Multiplication of Bignums:** Multiplication of bignums is perhaps the most critical function of the entire library. Modular exponentiation (the cornerstone of most public-key cryptosystems) makes extensive use of it. It must therefore be done as efficiently as possible. Knuth [Knuth 80] refers to many different methods for doing multiplication. The most obvious way is the traditional $O(n^2)$ method where n is the number of chunks in the bignum. Other methods are discussed which have potential gains when n becomes very large. For our purposes where n is likely to range from 4 (128 bits) to 64 (2048 bits), the Karatsuba $O(n^{log3})$ multiplication algorithm seemed promising. This algorithm is a recursive algorithm which divides the multiplicands into two equal length halves. The $O(n^2)$ multiplication of two numbers of length n is converted into 3 multiplications of numbers of length $n/2$ and some subtractions, additions and shifts. That is, two numbers A and B each of length $2N$ are multiplied as follows:

$$A*B = (2^N A_{hi} + A_{lo}) * (2^N B_{hi} + B_{lo})$$

$$A*B = 2^{2N}(A_{hi}B_{hi}) + 2^{N}(A_{hi}B_{hi} + A_{lo}B_{lo} + (A_{hi}-A_{lo})*(B_{lo}-B_{hi})) + A_{lo}B_{lo}$$

Shifts are avoided by dropping the subproducts, sums and differences in the appropriate places in the product array. This requires a temporary array for subproducts.

It is important to know where to stop the recursion and do a regular $O(n^2)$ multiplication of partial products. This number varies to some degree as a function of the compiler and computer architecture used. As the efficiency of the 32 bit primitive multiplication operation improves, the point at which recursion gives a noticeable improvement increases. That is, improvements due to recursion becomes less noticeable. We provide three 32 bit primitives: in-line assembly routines (for some architectures), Karatsuba 32 bit routine, and the standard n^2 method. These are given in decreasing efficiency. Using the assembly routine, the Karatsuba cutoff for a Sparc II is set at 4. Using the Karatsuba or n^2 methods, the cutoff is set at 2, or pure Karatsuba recursion. Tables 1 and 2 illustrate these results.

Since the algorithm works by splitting the numbers to be multiplied in half and doing smaller multiplications and then adding the pieces, it is most efficient if the length of the original numbers is the same and a power of 2. To make the routine general purpose we take the largest part of the smaller of the two multiplicands which is a power of 2 in length and do the recursive multiply on the parts of the numbers which are that length. The upper parts of the numbers not included in the recursive multiply are then included by an $O(n^2)$ routine which generally has only to do a small amount of work to complete the product of the original numbers. Tables 1 and 2 illustrate an example of this using 768 bit numbers ($768=24*32$ and 24 is not a power of 2).

This algorithm had been tested by some other groups [Bong 89] and the result had been that only for numbers of length greater than 320 bits would this algorithm give appreciable speedup. As we see in Tables 1 and 2 our implementation yields the similar results. Since it is likely that numbers greater than 320 bits in length will indeed be used, this is the method for doing multiplication that we have chosen for the library.

For squaring there is a well known speedup making use of the symmetry in the multiplication matrix which gives a 25% speedup. We have included this as well.

Karatsuba cutoff	128 bits	256 bits	512 bits	768 bits	1024 bits
multiply: 2	.064ms	.13ms	.40ms	.94ms	1.29ms
square: 2	.028ms	.10ms	.33ms	.85ms	0.97ms
multiply: 4	.038ms	.099ms	.33ms	.84ms	1.02ms
square: 4	.021ms	.075ms	.25ms	.76ms	0.79ms
multiply: 8	.038ms	.12ms	.33ms	.84ms	1.04ms
square: 8	.022ms	.07ms	.23ms	.74ms	0.75ms
multiply: 16	.038ms	.12ms	.42ms	.89ms	1.21ms
square: 16	.022ms	.07ms	.26ms	.76ms	0.82ms
multiply: 32	.038ms	.12ms	.42ms	.90ms	1.58ms
square: 32	.022ms	.07ms	.26ms	.57ms	0.98ms

TABLE 1. Bignum multiplication for Sparc II with inline asm 32 bit multiply primitive.

Karatsuba cutoff	128 bits	256 bits	512 bits	768 bits	1024 bits
multiply: 2	.11ms	.28ms	.82ms	2.88ms	2.63ms
square: 2	.06ms	.22ms	.63ms	2.62ms	2.07ms
multiply: 4	.08ms	.27ms	.85ms	2.95ms	2.92ms
square: 4	.07ms	.18ms	.58ms	2.67ms	2.02ms
multiply: 8	.10ms	.33ms	1.00ms	3.22ms	3.60ms
square: 8	.05ms	.20ms	0.62ms	2.77ms	2.23ms
multiply: 16	.10ms	.35ms	1.28ms	3.48ms	4.95ms
square: 16	.05ms	.20ms	0.75ms	2.87ms	2.72ms
multiply: 32	.08ms	.33ms	1.33ms	3.57ms	5.88ms
square: 32	.05ms	.20ms	0.77ms	2.15ms	3.30ms

TABLE 2. Bignum multiplication for Sparc II with Karatsuba 32 bit multiply primitive.

2.1.3 **Modular Exponentiation:** Exponentiation is done most efficiently using addition-chaining [Knuth 82 and Brickell 89]. Consider the modular exponentiation

$$Y = \alpha^{\beta}\ mod\ \delta$$

If b is a bignum of length L (L 32 bit unsigned longs), it is made up of at most $8*L$ 4-bit nibbles. For example, if $\beta = 0xABCD$, $L = 1$ and the nibbles are $0xA$, $0xB$, $0xC$, and $0xD$. The above equation can in this case be rewritten:

$$Y = ((((\alpha^{A})^{16} * \alpha^{B})^{16} * \alpha^{C})^{16} * \alpha^{D})(mod\ \delta).$$

Since the partial results are always of the form α^{i} where $0 \le i \le 15$, we calculate and store them as follows.

$$Y_0 = 1$$
$$Y_i = (\alpha * Y_{i-1})\ mod\ \delta$$
$$\text{for } 0 < i < 16$$

The value of each 4 bit nibble in the exponent β is used as an index into the array of the stored partial results. The partial results are calculated modulo δ to keep the size of the partial results reasonable. Raising each nibble's result to the 16th power is done as 4 squarings each followed by a modular reduction. The standard method for doing a modulo operation is too slow for large integers. It is widely accepted that the most efficient algorithm for doing modular reduction *many times using the same modulus* is Montgomery's method. [Mont 85 and Dussé 85]. This is the method that we use.

A more radical speedup in modular exponentiation can be achieved if one knows the factors of the modulus n. In RSA cryptosystems n is assumed to be the product of two large primes p and q (both of which are half the length of n). One can calculate $A^{B} mod\ n$ very quickly from $(A\ mod\ p)^{Bp} mod\ p$ and $(A\ mod\ q)^{Bq} mod\ q$ where $Bp = B\ mod\ p-1$ and $Bq = B\ mod\ q-1$. The Chinese Remainder Theorem is used to combine the two partial results. Tables 3 and 4 summarize our modular exponentiation results. (For the Sparcstation there is a time for doing exponentiation using the standard modulus operation (instead of Montgomery reduction) for the partial products. The gain using Montgomery's technique is obvious).

Algorithm	512 bits	1024 bits	Machine
Standard Modulus	5.3s	-	Sparc II
Montgomery Reduction	1.4s	10.3s	Sparcs II (N^2 32 bit mult)
Montgomery Reduction	1.2s	8.9s	Sparcs II (Karatsuba 32 bit mult)
Montgomery Reduction	0.43s	3.0s	Sparcs II (Assembly 32 bit mult)
Montgomery Reduction	1.0s	7.2s	Sparc 10 (Karatsuba 32 bit mult)
Montgomery Reduction	0.13s	1.05s	Sparc 10 (Assembly 32 bit mult)
Montgomery Reduction	0.12s	0.78s	R4000 Indigo (Assembly 32 bit mult)
Montgomery Reduction	2.9s	19.8s	Vax 8650 (N^2 32 bit mult)
Montgomery Reduction	0.39s	3.0s	NCR 486/50 (Assembly 32 bit mult)
Montgomery Reduction	3.1s	-	Macitosh IIci (Assembly 32 bit mult)

TABLE 3. $a^b mod\ c$ with a, b, c the same length.

Algorithm	512 bits	1024 bits	Machine
Chinese Remainder	0.44s	-	Sparc II (Karatsuba 32 bit mult)
Chinese Remainder	0.15s	0.95s	Sparc II (Assembly 32 bit mult)
Chinese Remainder	0.30s	1.90s	Sparc 10 (Karatsuba 32 bit mult)
Chinese Remainder	0.06s	0.40s	Sparc 10 (Assembly 32 bit mult)
Chinese Remainder	0.09s	0.57s	R4000 Indigo (Karatsuba 32 bit mult)
Chinese Remainder	0.04s	0.25s	R4000 Indigo (Assembly 32 bit mult)
Chinese Remainder	1.3s	7.3s	Vax 8650 (N^2 32 bit mult)
Chinese Remainder	0.14s	-	NCR 486/50 (Assembly 32 bit mult)
Chinese Remainder	1.8s	-	Macitosh IIci (Assembly 32 bit mult)

TABLE 4. $a^b mod\ c$ with a, b, c the same length and c the product of 2 primes.

2.1.4 Extended Euclid Algorithm:

Euclid's extended algorithm for finding the greatest common divisor of two numbers is based on the following equation:

$$aa' - bb' = gcd(a,b)$$

If a and b are relatively prime, their gcd is 1. In this case, if one can satisfy the above equation, then the multiplicative inverse, a', of $a(mod\ b)$ is known (and so is the negative multiplicative inverse, b' of $b(mod\ a)$). One place where this tool is useful is in generating keys for the RSA public key cryptosystem. If the public exponent, e and the modulus n ($=p*q$) are known and e is relatively prime to $\phi=(p-1)*(q-1)$, the private exponent d is the multiplicative inverse of $e(mod\ \phi)$. Euclid's algorithm is used to calculate it. Given a and b, we return a',b' and $gcd(a,b)$.

2.2 Random Number Generation

In cryptographic applications, it is often the case that random bits are required. These are used for session keys, exponents, prime generator seeds and stream ciphers. Random number generators generate either truly random bits or pseudorandom bits. The function which generates truly random bits should never cycle, its initial and current state should be indecipherable, its future states unpredictable from any collection of previous data and the output should be uniformly random (any one sequence is just as likely as any other). Generators with these qualities are considered to be cryptographically secure random bit sources.

Pseudorandom number generators are usually based on some complex function of the current state. The period of the function should be maximal, its output uniformly random and it should be implementable in an efficient manner. In addition to these properties, pseudorandom number generators to be used cryptographically must also be immune to cryptanalytic attacks. That is, output must yield very little information about subsequent output and the current state of the generator and the initial state should be secret or seeded with truly random bits which are discarded after initialization. The number of bits considered necessary for this initialization seed is whatever is computationally infeasible to exhaustively search in a reasonable amount of time. If the seed is changed frequently enough then fewer bits may be necessary. IETF suggests that 200 bits should be adequate [IETF].

As is discussed in the IETF randomness requirements document [IETF], getting absolutely secure, truly random bits in a way which is efficient and portable is nearly impossible except by means of some hardware device resident in all machines. We have included an implementation of a generator which we believe gives very random bits and which runs in a fairly common environment. We now describe this "truly" random number generator and a nonlinear feedback shift register pseudorandom number generator. We include results of chi-square tests [Knuth 80] for output from the generators. Values of chi between 0.1 and 0.9 indicate suitable randomness. Pseudo-random number generators were used to create 100 one megabyte files of random bits. Each file was tested and the results averaged to check for failure of particular chi-square tests. Our true random generator was used to create 1 and 5 megabyte files (these take half a day to several days on a Sparc II).

2.2.1 **Truerand()** Our truly random number generator, *truerand*() works by setting an alarm and then incrementing a counter register rapidly in the CPU until an interrupt occurs. The contents of the register are then exclusive-or'ed with the contents of an output buffer byte (truncating the register's data to 8 bits). After each byte of the output buffer is filled, the buffer is further processed by doing a right, circular shift of each character by 2 bits. This has the effect of moving the most active (and random) least significant bits into the most significant positions. This entire process is then repeated 3 times. Finally each character of the buffer has been touch by the two most random bits of the counter register after interrupts. That is, $4*n$ interrupts have occurred where n is the number of desired random bytes.

This generator obviously depends on the randomness of system interrupts and the granularity of the clock. On a very quiet system one would not expect the output to be very random. We assume a 60Hz clock and set the alarm at 16665 microsecs (1 tick). This implementation works on most machines running Unix System V, SunOS and BSD Unix. The chi-square results in table 5 show suitable randomness on heavily used workstations on busy networks. Tables 6 and 7 show the chi-square results on a very quiet NCR 486. These tables still show a significant degree of randomness. We can only assume that truerand() is very sensitive to whatever interrupts occur. This machine was not completely isolated.

Truerand() on a normally loaded machine

Tests using 1 Megabyte test files

A. Frequency Test	
B. Serial Test	CHI = 0.381038
C. Gap Test	CHI = 0.685569
D. Poker Test	CHI = 0.663350
E. Coupon Test	CHI = 0.365447
F. Permutations Test	CHI = 0.338432
G. Runs Up Test	CHI = 0.841052
H. Max-of-8 Test	CHI = 0.593779
I. Lapped M-Tuple Test	CHI = 0.294260

TABLE 5

Truerand() on a very lightly loaded machine

Tests using a 1 Megabyte test file	
A. Frequency Test	CHI = 0.790080
B. Serial Test	CHI = 0.287570
C. Gap Test	CHI = 0.278006
D. Poker Test	CHI = 0.774203
E. Coupon Test	CHI = 0.773437
F. Permutations Test	CHI = 0.475713
G. Runs Up Test	CHI = 0.038718
H. Max-of-8 Test	CHI = 0.528882
I. Lapped M-Tuple Test	CHI = 0.737021

TABLE 6

Truerand() on a very lightly loaded machine

Tests using a 4096000 byte test file	
A. Frequency Test	CHI = 0.191917
B. Serial Test	CHI = 0.902341
C. Gap Test	CHI = 0.568251
D. Poker Test	CHI = 0.548475
E. Coupon Test	CHI = 0.236782
F. Permutations Test	CHI = 0.358746
G. Runs Up Test	CHI = 0.741957
H. Max-of-8 Test	CHI = 0.035583
I. Lapped M-Tuple Test	CHI = 0.268988

TABLE 7

2.2.2 **Linear Feedback Shift Register Generators** Knuth discusses a random number generator attributed to Lewis and Payne [LewPay 73] which makes use of the following linear recurrence relation:

$$X_n = (X_{n-24} \oplus X_{n-55}) \; mod \; m, \; n > 55.$$

The generator can be efficiently implemented as a feedback shift register. The register has 55 (X_1 through X_{55}) elements in it and after each computation, X_{n-55} is replaced by X_n and the pointers into the register are incremented. The values 24 and 55 are not arbitrary but are chosen to yield maximum period. Knuth gives a table of these magic pairs for different sized registers [Knuth 80, p. 28]. The period of this function is known to be $2^{55} - 1$ for $m = 2^e$ [Tau 65]. As can be seen from table 7 and 8, the output is very sensitive to the initialization of the register; random initialization leads to random output. (*rand()* is a multiplicative congruential generator found with most releases of Unix ™. It is known to produce output which is not very random, particularly when the low order bits are used. However, since the output is related to the register contents, it is possible to know the contents of the entire register after it (the register) has completed a complete cycle through its elements. From this information, one can predict future output.

The Linear Feedback Shift Register Method Initialized by rand()

All bits 0-31 used from unsigned longs returned by RNG

Average statistics from 100 tests using 1 Megabyte test files

A. Frequency Test	CHI = 1.000000
B. Serial Test	CHI = 1.000000
C. Gap Test	CHI = 1.000000
D. Poker Test	CHI = 0.500888
E. Coupon Test	CHI = 0.485830
F. Permutations Test	CHI = 1.000000
G. Runs Up Test	CHI = 1.000000
H. Max-of-8 Test	CHI = 1.000000
I. Lapped M-Tuple Test	CHI = 0.496432

TABLE 8

The Linear Feedback Shift Register Method Initialized by truerand()

All bits 0-31 used from unsigned longs returned by RNG

Average statistics from 100 tests using 1 Megabyte test files

A. Frequency Test	CHI = 0.506955
B. Serial Test	CHI = 0.537025
C. Gap Test	CHI = 0.710635
D. Poker Test	CHI = 0.413936
E. Coupon Test	CHI = 0.521240
F. Permutations Test	CHI = 0.501915
G. Runs Up Test	CHI = 0.423917
H. Max-of-8 Test	CHI = 0.522343
I. Lapped M-Tuple Test	CHI = 0.419101

TABLE 9

2.2.3 **Cryptographically Strong Generators** There are other methods of generating random bits which are considered safe but for which efficient implementations are hard to realize. Rueppel discusses some of these in [Rueppel 92]. One makes use of RSA [RSA 78]. In this particular case, each bit requires a modular exponentiation. Even with very efficient implementations of modular exponentiation such a generator would be very inefficient.

The Data Encryption Standard [DES 77] used in output feedback mode is often mentioned as a cryptographically strong random stream generator. Shepp has shown that for random permutations, the average cycle length is on the order of n [Shepp 66], which in the case of DES is 2^{64}. The state of DES is at most 64 bits if only the initial block is truly random. If the key is also random then the random state is 120 bits. This is probably adequate for most applications.

We now discuss generators that make use of the confusion property of DES but have longer periods and more truly random state.

2.2.4 **A Nonlinear Feedback Shift Register Random Number Generator** Golomb showed that replacing linear feedback logic of feedback shift register generators with nonlinear logic can result in a considerable increase in the generator period [Gol 67]. It is also probable that using an appropriate nonlinear function, the state of the register's confusion will become greater, thereby increasing its cryptographic strength. (Confusion here means that every element in the register is a complex transformation of every other element in the register [Rueppel 92]).

Our generator is very similar to the linear feedback shift register generator discussed previously. We replace the exclusive-or operation in Lehmer's function by a nonlinear permutation function of the concatenation of the two elements X_{lower} and X_{upper}. The most significant half of the permutation replaces X_{upper} and the least significant half is output, *thereby ensuring independence of the register contents and the returned sequences.*

The nonlinear permutation we use is the DES [DES 77] block cipher function. As discussed above, DES has a maximum cycle on the order of 2^{64}. However, since we have a register which is N X 32 truly random bits (N is the register length) and a truly random DES key, the generator period is that of a random mapping or a constant times $(2^{N*32}-1)^{1/2}$ [Fla 89]. If instead of replacing X_{upper} with the most significant half of the output of DES, we exclusive-or the contents of X_{upper} with the most significant half of the DES output, the expected longest period becomes that of a random permutation, a constant times $(2^{N*32}-1)$.

We have therefore:

$$m = 2^e,$$

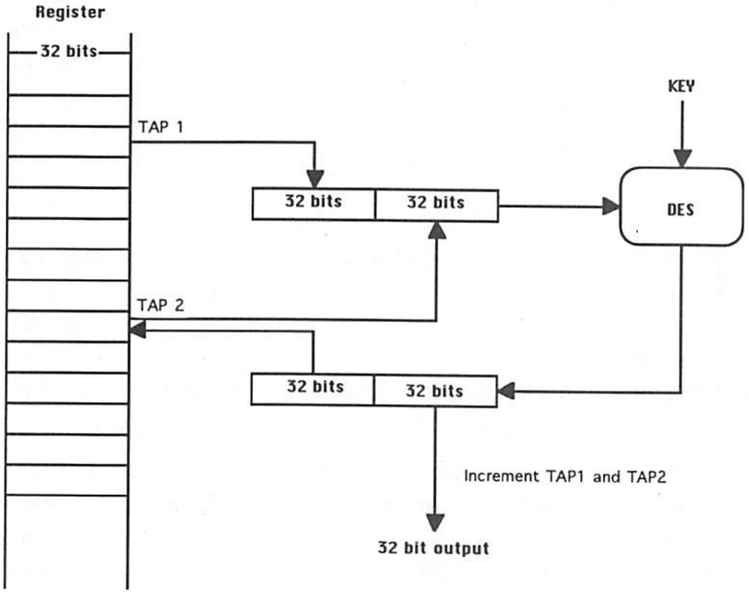
$$Y_n = DES((X_{lower} << 32) \mid X_{upper}, KEY),$$

$$OUT = Y_n \ mod\ m,$$

$$newX_{upper} = Y_n >> e$$

$$X_{upper} = X_{upper} \oplus newX_{upper}.$$

Figure 1 depicts this function for $e = 32$.



A Non-linear Feedback Shift Register Random Number Generator

Figure 1

2.2.5 **Implementation** We implement the register as an array of 7 unsigned longs. These are initialized with 7 unsigned longs from a static table generated by our truly random generator. The table is then further randomized by a vector of 28 truly random bytes either generated upon the first call to the generator or supplied by the user. These bytes are exclusive-or'ed into the table. The generator then calls itself $7 * \log_2 7$ times so that each register entry becomes a function of every other entry. Last, the register pointers X_{lower} and X_{upper} are set at entries 3 and 7, respectively. These are the "magic" values for a register of length 7 [Knuth 80]. After this point the generator is completely initialized with 224 bits of truly random state.

When the generator is called after initialization, the elements pointed to by static pointers X_{lower} and X_{upper} are concatenated to form a 64 bit array. This is then passed to the *DES* block encryption function together with a static, randomly initialized, 8 byte key. The encrypted (permuted) output is then treated as previously discussed. Finally, X_{upper} and X_{lower} are incremented. For either pointer, when the end of the register is reached, that pointer is reset to the first register element.

This implementation generates random bits at 0.4 Megabits/second on a Sparcstation II. The following tables illustrate its randomness characteristics.

The Nonlinear Feedback Shift Register initialized by truerand()

Bits 0-31 used from unsigned longs returned by RNG

Average statistics from 100 tests using 1 Megabyte test files

A. Frequency Test	CHI = 0.499961
B. Serial Test	CHI = 0.543879
C. Gap Test	CHI = 0.538618
D. Poker Test	CHI = 0.473287
E. Coupon Test	CHI = 0.515773
F. Permutations Test	CHI = 0.474690
G. Runs Up Test	CHI = 0.485048
H. Max-of-8 Test	CHI = 0.519509
I. Lapped M-Tuple Test	CHI = 0.504154

TABLE 10

The Nonlinear Feedback Shift Register initialized by truerand()

Bits 0-31 used from unsigned longs returned by RNG

Average statistics from 5 tests using 50 megabyte test files

A. Frequency Test	CHI = 0.442733
B. Serial Test	CHI = 0.458336
C. Gap Test	CHI = 0.348575
D. Poker Test	CHI = 0.525462
E. Coupon Test	CHI = 0.464458
F. Permutations Test	CHI = 0.286269
G. Runs Up Test	CHI = 0.555994
H. Max-of-8 Test	CHI = 0.296056
I. Lapped M-Tuple Test	CHI = 0.572037

TABLE 11

2.3 DES

The implementation of the DES algorithm was developed in 1982 and is based on table look-up. DES operates by the repeated process of hashing one half of a block and XOR'ing that with the other half. The 32-bit half-block hash is performed by splitting up the word into eight 6-bit pieces. These are XOR'ed with some key bits, then used to index combined S-and-P look-up tables, giving eight 32-bit pieces. Those 32-bit pieces are OR'ed together to give the hash value. The key is cached to avoid key setup for every block cipher operation using the same key.

We provide tools for doing basic DES encryption and decryption in electronic codebook, cipher block chaining and output feedback modes. In ECB mode, DES runs at 0.9 Megabits/second on an NCR 486/50 machine. Similar speeds are achieved on a SUN Sparc II.

2.4 Primes

To generate random prime numbers we generate a random odd number which is the length we seek. We then test it to see whether it is prime using two tests. First the candidate is passed through a test which determines whether it is divisible by one of the first 54 primes (2 - 251).

These primes are chosen as they are the primes which will fit into one byte making a very efficient calculation of the modulus possible [Gordon 84]. Rabin's probabilistic primality test [Knuth 80] is then used. Rabin's test gives a probability of $1/4^n$ that the number is not prime where n is number of passed tests. Gordon suggests that choosing n to be 5 is adequate [Gordon 84]. The NIST proposal for digital signatures states that n should be 50. It is almost invariably the case that non-primes fail Rabin's test on the first pass. It is also the case that probable primes which are not prime are very rare. For these reasons and because each pass through Rabin's test involves a modular exponentiation (which is time consuming), we side with Gordon and use $n = 5$. (We do however provide to the user a means to change this value). If a test fails, we add 2 to the candidate and try again until a candidate passes n probabilistic tests.

It is often the case that primes which are considered strong are desired. These are primes with at least the property that the prime minus 1 has a prime factor, say f. That is, $(p-1)\ mod\ f=0$ where p and f are prime. Factoring very large numbers which are products of primes is one of the hardest known problems and it is this difficulty which gives some cryptosystems their strength. There is some evidence that using strong primes increases the difficulty of the factoring problem. To find this set of primes $\{p,f\}$ we use Gordon's method [Gordon 84] which includes the above restriction and several others and adds only a small overhead to the general problem of finding random primes. We also include the prime generation method suggested by NIST for use with the Digital Signature Standard [NIST 91].

Timing results for finding Gordon strong primes are summarized in Table 12. In these tables, the results are the average times for 100 primes. There is quite a bit of variation in the time that it takes to find a prime from a random starting point.

256 bits	512 bits	768 bits	1024 bits	Machine
2.8s	24.0s	2.0m	5.1m	Sparc II (assembly 32 bit multiply)
0.5s	4.6s	40.0s	1.7m	Sparc 10 (assembly 32 bit multiply)
0.7s	6.3s	32.0s	1.1m	R4000 Indigo (assembly 32 bit multiply)
2.3s	22.0s	2.0m	-	NCR 486/50 (assembly 32 bit multiply)
21.2s	3.3m	-	-	Vax 8650 (n^2 multiply)

TABLE 12. Strong prime generation average times for 100 primes and 5 passes of Rabin's test.

2.5 Message Digests

For many cryptosystems a one-way hashing function is needed which will take an arbitrary amount of input and yield a message digest which is impossible to generate with any message other than the one used to create the digest. For this purpose we use the MD5 message digest functions from RSA Data Security, Inc. [MD5 91] and the NIST secure hash standard (SHS) [NIST 92]. We provide a *Bignum* interface to these functions.

2.6 RSA Tools

The basic RSA cryptosystem assumes two key exponents e and d and a modulus n which is the product of two strong primes p and q. The security of RSA is based on the difficulty of factoring n. The exponents are chosen such that $e*d=1(mod\ \phi)$ where $\phi=(p-1)*(q-1)$. A message m is encrypted by computing

$$E(m)=m^e mod\ n$$

and the result is decrypted by computing

$$m=D(E(m))=E(m)^d mod\ n.$$

Normally e, the public exponent, is chosen to be a small random number relatively prime to $phi(n)$. d, the private exponent, is then calculated from e, p and q using Euclid's extended greatest common divisor algorithm. d is usually the same length as the modulus.

We provide the user with a key set generation tool which takes the modulus size and the public exponent size as arguments. The key set contains the public key $\{e \text{ and } n\}$ and the private key $\{d,p,q,d \ mod \ (p-1),d \ mod \ (q-1),c12\}$. The private key is set up for the Chinese Remainder Theorem speedup discussed above ($c12=p^{-1} \ mod \ q$).

Times for RSA encryption, decryption, signing and signature verification are given in table 13.. The times for signing assume that a hash of the document to be signed has already been formed. The signature is then really just RSA decryption of the hash using the private key.

Operation	512 Bits	768 Bits	1024 Bits	Machine
Encrypt	0.03s	0.05s	0.08s	Sparc II
Decrypt	0.16s	0.48s	0.93s	Sparc II
Sign	0.16s	0.52s	0.97s	Sparc II
Verify	0.02s	0.07s	0.08s	Sparc II
Encrypt	0.01s	0.02s	0.03s	Sparc 10
Decrypt	0.08s	0.22s	0.40s	Sparc 10
Sign	0.08s	0.22s	0.40s	Sparc 10
Verify	0.01s	0.03s	0.04s	Sparc 10
Encrypt	0.007s	0.01s	0.02s	R4000
Decrypt	0.04s	0.13s	0.25s	R4000
Sign	0.04s	0.13s	0.25s	R4000
Verify	0.007s	0.01s	0.02s	R4000

TABLE 13. RSA function times for different modulus lengths with public exponents assumed to be 8 bits.

2.7 El Gamal Tools

El Gamal assumes a strong prime p, a number α (a primitive root $mod \ p$), a random, private secret *secret* and public $Y = \alpha^{secret} mod \ p$. The security of this system depends on the difficulty of doing discrete logarithms for large numbers. To encrypt a message m, a user Alice, holding a public key Y_b of an intended receiver Bob, generates a random number x and a key $K = Y_b{}^x mod \ p$. The encrypted message is then sent as the pair $\{c1, \ c2\}$ where $c1 = \alpha^x mod \ p$ and $c2 = K*m \ mod \ p$. Bob (who holds the the private key *secret* corresponding to Y_b) then recovers the key $K = c1^{secret} mod \ p$ and calculates $m = c2*K^{-1} mod \ p$.

Similar games may be played to compute digital signatures for messages. Assume $0 < m < p-1$. A digital signature (generated by Alice) for a message m is the pair (r,s) where r and s are defined such that

$$(1) \ \ \alpha^m mod \ p = ((Y_a{}^r mod \ p)*(r^s mod \ p)) \, mod \ p$$
$$(2) \ \ r = \alpha^k mod \ p$$

where k is random, chosen such that $0 < k < p - 1, \ \ gcd(k, \ p - 1) = 1$.
Expression (1) can then be rewritten:

$$(3) \ \ \alpha^m mod \ p = \alpha^{X_a * r + k * s} mod \ p$$

so that

$$(4) \ \ m = (X_a \ * \ r \ + \ k \ * \ s) \ mod \ (p \ - \ 1).$$

From (4) and the fact that $gcd(k,p-1)=1$, s can be solved for. Alice sends Bob m,and(r,s). To verify that (r,s) is indeed Alice's signature of m, Bob must verify that (1) holds for m,r and s.

We provide the following tools for El Gamal systems: a key set generation tool which returns a public and private key, message encryption and decryption tools, a signature generation tool and a signature verification tool. Times for encryption, decryption, signing and signature verification are in the following table. Again, signature times assume a hash value for the document to be signed has been formed and the signature is computed for that value.

Operation	512 Bits	768 Bits	1024 Bits	Machine
Encrypt	0.33s	0.80s	1.09s	Sparc II
Decrypt	0.24s	0.58s	0.77s	Sparc II
Sign	0.25s	0.47s	0.63s	Sparc II
Verify	1.37s	5.12s	9.30s	Sparc II
Encrypt	0.12s	0.30s	0.23s	Sparc 10
Decrypt	0.10s	0.23s	0.20s	Sparc 10
Sign	0.10s	0.18s	0.13s	Sparc 10
Verify	0.58s	2.00s	2.55s	Sparc 10
Encrypt	0.08s	0.20s	0.30s	R4000
Decrypt	0.07s	0.14s	0.21s	R4000
Sign	0.05s	0.13s	0.20s	R4000
Verify	0.36s	1.26s	2.36s	R4000

TABLE 14. El Gamal function times for different modulus lengths and 160 bit exponents.

2.8 The NIST Digital Signature Algorithm (DSA):

The National Institute of Standards and Technology (NIST) has patented a digital signature algorithm (DSA) which is based on El Gamal but which places restrictions on the key parameters which are different from those described above [NIST 91]. Times for signature generation and verification for several architectures are given in the following table. The same assumptions about signature times as were made for RSA and EL Gamal are also made here.

Operation	512 Bits	768 Bits	1024 Bits	Machine
Sign	0.20s	0.43s	0.57s	Sparc II
Verify	0.35s	0.80s	1.27s	Sparc II
Sign	0.08s	0.13s	0.22s	Sparc 10
Verify	0.12s	0.30s	0.40s	Sparc 10
Sign	0.05s	0.11s	0.14s	R4000
Verify	0.09s	0.21s	0.29s	R4000

TABLE 15. NIST DSA times for different modulus lengths and 160 bit exponents.

3. Applicability and Protocol Issues

In order for software implementations of cryptosystems to be useful in particular services, the overhead imposed by their use must be not be great enough to break or severely degrade the services. In particular, network protocols sensitive to subtle timing of message exchange must not have that timing altered so as to break the protocols. Crypto-protocols involving public key systems need to execute with speed which is consistent with the protocol application.

The fastest known hardware implementation of DES runs at 1 Gigabit/second [Eberle]. This is fast enough for some video applications as well as for fast exhaustive search for DES keys (the claim is 16 days for $\$1M$ in DES chips (estimated at $300 each)). The fastest known hardware implementation of RSA runs at 1 Megabit/second (decryption using 512 bit modulus) [Shand 93]. It is clear that using such implementations will impose very little overhead in most applications. However, these implementations are not cheap and cannot be expected to be ubiquitous.

Our implementation of DES (1 Megabit/second on a Sparc II) is useful in a number of applications. 1 Megabit/second DES can be used for real-time encryption of T1 lines (64kbps) and 64kbps digital speech. Using 1 Megabit/second DES in a file system level encryption system [Blaze 93] one sees a performance penalty of around a factor of 2. For *swIpe*, a network layer encryption application [Ioannidis 93] which uses this implementation of DES, it is clear that 1 Megabit/second is not efficient enough to avoid network throughput degradation.

The public key systems implemented in CryptoLib (RSA runs at 3.2 kilobits/sec on a Sparc II with 512 bit modulus) are useful for most non-realtime applications in which delays on the order of a few seconds are tolerable. Implementation of a Key Certification Authority for off-line public key certificates is easily done in software. There are many examples of systems in which large documents are digitally signed, encrypted and then stored periodically for some accounting purpose. This too is easily accomplished using software implementations of public key systems. *swIpe* uses an implementation of RSA (RSA-REF's) for key exchange but no penalty is observed as key exchange happens infrequently. Another example of key exchange protocols being implemented in software occurs in telephone security devices in which the participants are willing to pay for a few seconds of overhead in exchange for the guarantee of privacy.

Whether or not software speeds for public key protocols such as Diffie-Hellman key exchange (in which each side of the exchange must do two modular exponentiations for prime moduli of least 768 bits) will be good enough for applications which are extremely time sensitive (such as cellular telephony) remains an open question. There are protocols which make use of modular squaring and square roots to reduce the amount of overhead in the crypto-protocol. However, it has been shown that significantly greater execution speeds are possible for software implementations of modular arithmetic on digital signal processors [Dussé 85] so that full 768 bit modular arithmetic may be possible for devices containing DSPs. In fact, in many devices using DSPs, the DSPs are idle during call setup. So, rather than having special purpose hardware for crypto-protocols used for key exchange, it makes sense to use the DSPs and a software implementation.

4. Conclusion

We have shown that efficient implementations of algorithms needed for cryptosystem research are possible. Indeed, these implementations are efficient enough for many applications. The main point is that the portability of software more than compensates for overhead penalties in many cases. There are still many open research issues regarding timing and protocol implementation and what sort of overhead imposed by encryption and key exchange can be tolerated.

We are now planning a protocol front end to CryptoLib. There are many difficult issues here. What kinds of objects are being encrypted? Streams? Files? Keys? Video? What should the interface to encryption functions look like? How will trustworthy public keys be generated, stored and retrieved? How are keys mapped to different entities? How do questions like this affect the design of a protocol front end?

CryptoLib is an ongoing collection of implementations of algorithms necessary for the various crypto-protocols being designed to solve the many subtle problems of privacy and security as they arise.

We would like to thank Steve Bellovin, Matt Blaze, Michael Merritt, Andrew Odlyzko and Jim Reeds for their advice and continuing support of this work.

5. References

[Blaze 93]
M. Blaze. A Cryptographic File System for Unix. *Proc. 1st ACM Conference on Computer and Communication Security*, Fairfax, VA, November, 1993.

[Bong 89]
Dieter Bong and Christoph Ruland. Optimized Software Implementations of the Modular Exponentiation on General Purpose Microprocessors. *Computers and Security* 8,7 (1989), 621-630.

[Brickell 89]
Ernest F. Brickell. A Survey of Hardware Implementations of RSA. Viewgraphs from a Technical talk by Brickell at AT&T Bell Laboratories, Murray Hill, NJ, (Sept 12, 1989).

[DES 77]
Data Encryption Standard. *Federal Information Processing Standards Publication 46-1*

National Technical Information Service, U.S. Dept. of Commerce, 1977.

[Diffie 76]
W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory* IT-22,6 (Nov, 1976), 644-654.

[Dussé 85]
Stephen R. Dussé *et. al.* A Cryptographic Library for the Motorola DSP56000. *Advances in Cryptology - EUROCRYPT '90* I.B. Damgard ed., Springer Lecture Notes in CS #473, 1991, pp. 230-244.

[Eberle]
Hans Eberle. A High-speed DES Implementation for Network Applications. *SRC Research Report 90*, DEC Systems Research Center.

[ElGamal 85]
Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory* IT-31,4 (July, 1985), 469-472.

[Fla 89]
Philippe Flajolet and A. M. Odlyzko. Random Mapping Statistics, *Advances in Cryptology: Proceedings of Eurocrypt '89*, J. Quisquater, ed., Springer Lecture Notes in Computer Science, (1990), pp. 329-354.

[Golomb 67]
Solomon W. Golomb. *Shift Register Sequences,* San Francisco: Holden Day, 1967

[Gordon 84]
J. A. Gordon. Strong Primes are Easy to Find. *Proc. Eurocrypt-84*, Paris, April, 1984.

[IETF]
Internet Engineering Task Force Randomness Requirements for Security RFC, 3/25/93.

[Ioannidis 93]
John Ioannidis and M. Blaze. Architecture and Implementation of Network-Layer Security Under Unix. *Proc. USENIX Security Workshop*, Santa Clara, CA, October, 1993.

[Knuth 80]
Donald E. Knuth. *The Art of Computer Programming, Vol 2: Seminumerical Algorithms.* (Addison-Wesley, 1982, 2nd edn.)

[Lehmer 77]
D. H. Lehmer. *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery,* (Cambridge: Harvard University Press, 141-146, 1951). 1977.

[LewPay 77]
T. G. Lewis and W. H. Payne. *JACM* **20** *(1973), 456-468.*

[MD5 91]
RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright (C) 1990, RSA Data Security, Inc. All rights reserved. Received via the net.

[Mont 85]
Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation* 44,170 (April, 1985), 519-521.

[NIST 91]
National Institute of Standards and Technology (NIST). Digital Signature Standard. *Federal Information Processing Standards Publication XX* National Technical Information Service, U.S. Dept. of Commerce, 1991.

[NIST 92]
National Institute of Standards and Technology (NIST). Secure Hash Standard *Federal*

Information Processing Standards Publication YY National Technical Information Service, U.S. Dept. of Commerce, 1992.

[RSA 78]
R. L. Rivest, A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM* 21,2 (Feb, 1978), 120-126.

[Rueppel 92]
Rainer A. Rueppel. Stream Ciphers, *Contemporary Cryptology: The Science of Information Integrity,* edited by Gustavus J. Simmons, (IEEE Press, 65-134, 1992).

[Shand 93]
Mark Shand and J. Vuillemin Fast Implementations of RSA Cryptography. Digital Equipment Corporation, Paris Research Lab (PRL). *Technical talk at AT&T Bell Laboratories*, Murray Hill, NJ, (July 6, 93).

[Shepp 66]
L. A. Shepp and S. P. Lloyd Ordered Cycle Lengths in A Random Permutation *Trans. Amer. Math. Soc.*, 121 (1966), pp. 340-357.

[Tau 65]
R. C. Tausworthe. *Math. Comp.* **19** *(1965), 201-209.*

Long Running Jobs in an Authenticated Environment

A.D. Rubin
rubin@citi.umich.edu

P. Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
The University of Michigan
Ann Arbor

ABSTRACT

In strong authentication systems, users may obtain access to secure system resources only when in possession of valid credentials. These are issued with limited lifetimes; their renewal requires a user to enter her password. We have developed a system called khat with which a user may schedule a batch job to be run at a later date in the current environment. The batch job is stored on a secure machine, and sent and received in encrypted form. When it is time for the job to run, the server generates credentials for the originating user and sends them encrypted to the machine on which the job will run. The user is given an option to specify that tickets should be continually generated for the job until its execution has completed.

1. Introduction

Adaptations of Needham and Schroeder's authentication system [1] are a boon for establishing secure services in distributed systems. One such adaptation is the Kerberos Authentication system [2] of MIT's Project Athena. An unfortunate byproduct of building Kerberos-based systems is a loss of functionality, such as long running jobs. In this paper, we address this weakness and offer a solution.

Before Kerberos, UNIX authentication was coterminous with a login session. In the Kerberos system, tickets[1] expire, so that a compromised[2] ticket does not allow an imposter to masquerade as an authenticated user forever. Consequently, users are forced to reauthenticate on a regular basis, usually about once a day, to acquire fresh tickets.

For a Kerberos user to submit and execute a long-running batch job that employs secure system resources, she must either physically reauthenticate whenever tickets are about to expire, or enter a password into a script. Similarly, it is impossible to schedule a batch job to run at a distant future date and be authenticated as the user.

This problem has been recognized as a difficult one. Lampson *et al.* state that "it is a tricky exercise in balancing the demands of convenience, availability, and security" [3]. They further state that "the basic idea is to have a single highly available agent for the user that replaces the login workstation and refreshes credentials for long-running jobs." This approach is the one we take in solving the problem for Kerberos-based systems.

[1] Kerberos credentials.

[2] *E.g.,* stolen.

2. KHAT: A New Kerberos Service

This section describes a new service: `khat`, based on the UNIX `at` command.[3] In a later section, we critique the design of `khat` and offer suggestions for improvement.

Like `at`, which is used to schedule batch jobs at a specific time and date, `khat` offers a batch service. The principal difference between the two is that `khat` provides continuous Kerberos authentication to the batch job while it runs.

2.1. Overview of KHAT

When a user wishes to schedule a batch job, she issues the `khat` command with a syntax very similar to the UNIX `at` command. A spool file for the batch job is created, containing, among other things, the user's name, her current working directory, and her shell environment. So far, this is identical to `at`.

Next, `khat` requests a ticket for the `khat` service from the Kerberos ticket granting server on behalf of the user. This step is shown in Figure 1.

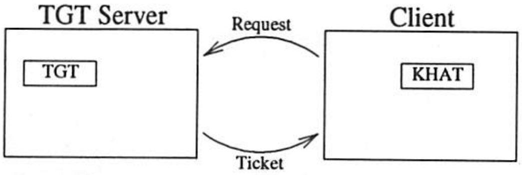


Figure 1. The client sends the ticket granting ticket (TGT) server a a request for a `khat` ticket. The response includes a ticket that enables the `khat` server to authenticate the client.

The `khat` ticket is then sent to a `khatd` server. The server must run on a secure machine. The client and server then mutually authenticate [2], as shown in Figure 2.

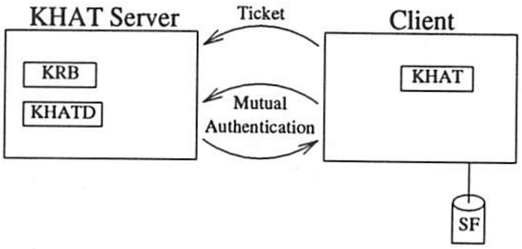


Figure 2. The client has generated a spool file (SF) for the `khat` job, which it stores on the local disk. It sends a `khat` ticket to the server (KHATD), and the client and server mutually authenticate. Note that the Kerberos server (KRB) is running on the same machine as `khatd`.

After mutual authentication, the client and server share a DES [4] key, which we denote K, as shown in Figure 3.

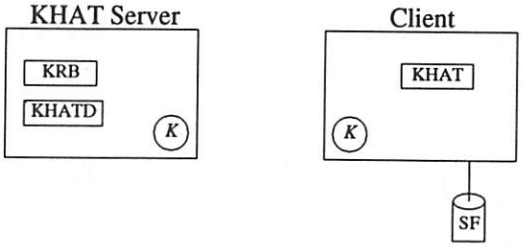


Figure 3. Initially, the spool file resides on the client machine. The client and server share a secret key, K.

The `khat` client uses K to seal the spool file. This hides the details of the batch request from prying eyes,

[3] The name is taken from Kerberos and `at`. The additional letter stems from a tradition that we don't completely understand, but slavishly follow.

as well as assuring its integrity. The encrypted spool file is then sent to the khatd server, as shown in Figure 4.

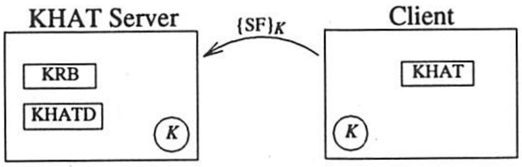


Figure 4. The client sends the spool file to the server, sealed under the session key, *K*.

Along with the spool file, the client sends information such as the time and date for the job to run, the user's environment, *etc*. The khatd server receives the spool file from the client, unseals it, and stores the file away until it is time for the job to run. At this point, the client discards its copy of the spool file and listens to a well known port for activation. For reasons discussed later, the khatd server runs on a Kerberos master or slave machine. This blocked configuration is depicted in Figure 5.

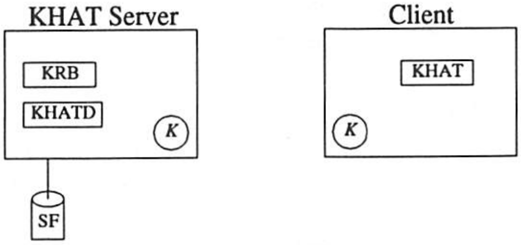


Figure 5. The spool file is stored on the secure server machine, and is purged from the client machine.

Periodically,[4] the server checks to see if any job is scheduled to be run. When the time for the job to run arrives, the server uses the Kerberos database to construct a ticket for the user. It then seals this ticket along with the spool file, and sends them back to the client machine, as shown in Figure 6.

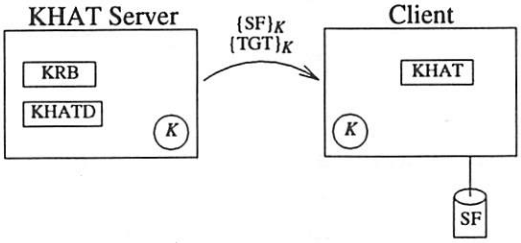


Figure 6. When it is time for the job to run, the encrypted spool file and a ticket granting ticket (TGT) are sent to the client under the session key.

The client can use the TGT to obtain tickets for other services. If the job will run for longer than the life of the ticket, or if the user suspects this may be the case, khat offers an option to renew tickets, in which case the server sends new tickets to the client as long as the job is running. Of course, care is taken to ensure that the job is still running. (This proves to be a difficult problem.)

After the job terminates, the spool file is removed from the client machine, and the khat process exits. A message is then sent to the server machine so that the khatd process can exit too. Details follow in Section 3.6.

2.2. Implementation of the khatd server

The server program, khatd, runs as root. Moreover, the server machine contains a master or slave copy of the Kerberos database; any program running on such a machine must be trustworthy. The khat program runs on the client's host machine. It also runs as root, but care is taken to make sure that the user's job is not able to obtain a higher privilege than it should. The program uses the UNIX setuid facility to ensure that the user's batch job runs as that user. However, the actual khat program runs as root because

[4] Once a minute, in our implementation.

it must perform operations that require special privileges, such as setting group IDs, deleting files, and copying files to and from the local disk.

When the khatd server needs to issue tickets for a user, it sends a request to the Kerberos server. The Kerberos server returns a user ticket encrypted under that user's secret key. Unfortunately, the client machine does not necessarily have a user authenticated to it when this happens, so the user's secret key may not be available. Thus, the user's ticket issued by the Kerberos server is not readable.

To decipher the user ticket, the khatd server uses the Kerberos database to access the user's secret key. User keys are stored encrypted under a master key in the Kerberos database, so khatd must first use the Kerberos master key. Once it has decrypted the user's secret key, khatd decrypts the user ticket received from the Kerberos server. Finally, khatd changes the client address in the ticket to that of the target host that will run the job. The ticket is then ready to be sent to the target host. The encryption of the ticket before it is sent over the network is the topic of Section 3.5.6.

When a user issues the khat command, the TGT is used to request a khat ticket. After mutual authentication, the client receives a ticket and can then communicate with the server. Thus, khat behaves as any other Kerberos service, with one exception: to construct the ticket, khatd has to access the Kerberos database and the Kerberos master key. Thus, khatd must run on a Kerberos master or slave machine.

3. The Theory Behind KHAT

It seems contradictory to provide authentication for an absent principal. By definition, authentication means that a principal proves her identity, seemingly an impossible task if she is no longer present. Thus, to schedule a long-running job, or one to be run at a later date, a principal must leave something around so that possession of the thing is equivalent to authentication for that principal. A similar idea, called delegation, is discussed by Lampson *et al.*[3]. The authors define the ''speaks for'' relationship and provide rigorous definitions and proofs based on a set of axioms they define in the paper.

Ideally, we would like the user's batch job to delegate authority to the workstation, saying that the workstation speaks for the user. In general, though, we are dealing with the domain of untrusted workstations. Many workstations reside in public sites where many different users have access to them at all times. It is a fundamental assumption that nothing on such a workstation can be trusted. However, some compromises must be made to provide for authenticated long-running jobs; we elaborate on this theme in the next section.

3.1. The authentication problem for a vacant workstation

We call a workstation *vacant* whenever a given user's task must be run there while that user is not logged in. In the previous section, we assumed that nothing on the workstation could be trusted. The reason for this is straightforward: we must allow for the possibility that a user might obtain root privileges, *e.g.*, by booting the machine into single-user mode, whereupon the privileged user might replace any or all utilities on the workstation, including the operating system image itself. The only objects on a public workstation safe from such attack are those that are encrypted. Yet, we take it as given that the encryption key may not reside on the workstation, even if well hidden.

Experts dismiss systems that hide cryptographic algorithms or protocols (a.k.a. ''security through obscurity''). Kahn [5] cites Kerckhoffs' classic treatise on military security [6]. Saltzer and Schroeder [7] reflect a more modern view in describing ''open design'' as one of the basic principles of information protection:

> The design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user may be allowed to convince himself that the system he is about to use is adequate for his purpose. Finally, it is simply not realistic to attempt to maintain secrecy for any system which receives wide distribution.

Voydock and Kent amplify this perspective: ''data encryption is the fundamental technique on which all communications security measures are based'' [8]. Cloaking information does not protect it.

Therefore, in a secure, distributed authentication system, data must travel across the network encrypted; for two peers to communicate this way, they must share a secret. Thus, the user must place something on the workstation which the server can later use for mutual authentication.

Lampson *et al.* describe a mechanism whereby a vacant workstation could share a secret [3]. Their method requires that a machine possess a private key stored in nonvolatile memory. In addition to the private key, certificates and other rules must be stored on the boot ROM.

Aside from the fact that our workstations do not contain this information in ROM, Lampson's method requires a public key system, which is not compatible with Kerberos. At some future date, it may be possible to authenticate a workstation, whereupon it will not be necessary for the client to leave anything on the machine.

3.2. How KHAT authenticates to a vacant workstation

For the server to send an encrypted spool file and tickets back to the client's host machine, some shared secret must be left on the workstation. The creation and responsibility of this secret is illustrated in Figure 7. To leave this secret on the workstation, a new random key is generated, which we denote *N*.[5] *K*, the `khat` session key, is encrypted with *N* and stored on the client machine. *N* is then sent encrypted under *K* to the server. Note the symmetry here: the client holds $\{K\}_N$ but sends the server $\{N\}_K$.

After sending the spool file and other information to the server, the client erases all of this information from disk and from memory. All that the workstation keeps is $\{K\}_N$, the session key from the original `khat` ticket encrypted under *N*, accessible only to root on the client machine.

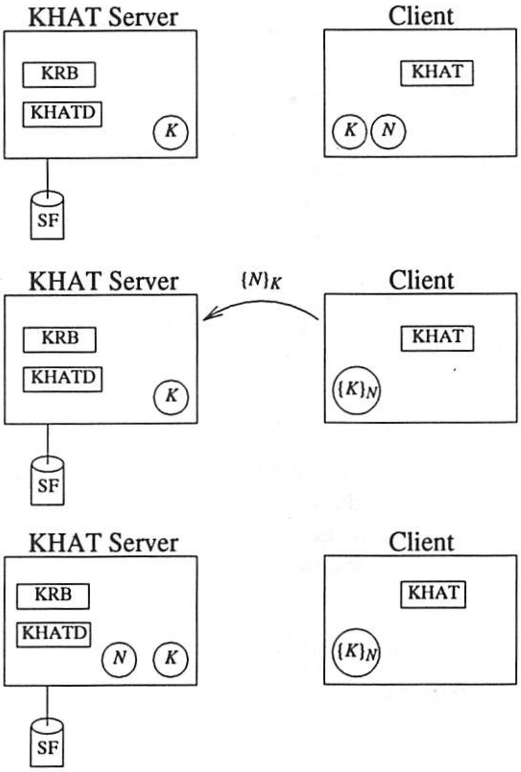


Figure 7: How *K* is secured on the client. The first step is illustrated in the top diagram. The client generates a random key, *N*. Then, as shown in the next diagram, *N* is used to encrypt the session key, *K*, which remains on the client machine. Then, *N* is sent to the server under the session key. Finally, as shown in the bottom diagram, *K* and *N* are stored on the server. When it is time for the job to run, *N* is sent to the client to unseal $\{K\}_N$.

When it is time for the job to run, the server sends *N* to the workstation in the clear, followed by the spool file and tickets, sealed under *K*. The workstation then unseals the original session key and uses it to decrypt the spool file and tickets.

[5] For *nonce*.

By itself, eavesdropping on the network does not expose the user: the only function served by N is to unseal the key on the workstation. Once the session key is unsealed, the spool file and tickets sent across the network can be decrypted by the `khat` agent sitting on the workstation.

3.3. Risks of running KHAT with a vacant workstation

In this section, we discuss the risks involved in running `khat` on a vacant workstation. If the workstation is rebooted, then the process memory is lost. Although a denial of service results, this can be reported back to the user and no real harm is done. If an imposter manages to gain control of the machine without erasing the memory, and examines memory to find the secret key, this will give no advantage, as the secret key is encrypted with N. In fact, a few other safeguards are in place. The imposter has no idea when a job is scheduled to run, as all such information has been kept secret and no longer resides on the workstation.

To compromise a job, an imposter must acquire the session key. But K is never exposed over the network, so the imposter must compromise the workstation itself. Even here, this must be accomplished either while the `khat` service is being requested, or while the user's job is being run. Even if $\{K\}_N$ is obtained from the workstation's process memory and N obtained through network eavesdropping, the imposter will be left with a ticket usable only on the target host. In summary, an imposter must completely compromise the workstation to affect the batch job. We know of no system that can run jobs securely on such a machine.

To deny service, an imposter can simply reboot a workstation, but `khat` can notify the user of unsuccessful batch jobs (soon to be implemented). As long as users understand this risk, they can choose whether to use the `khat` service. `Khat` does not compromise the security of people who do not use it.

3.4. The single user approach

The vacant workstation problem was addressed by Treese at MIT, where access to an Athena workstation is limited to one user at a time [9]. Treese reports that ''experience has shown that this is an acceptable limitation.'' Placed in our context, workstations could prohibit any login while any `khat` job is scheduled. This would make `khat` client machines much more secure. However, this would allow a denial of service attack, wherein a malicious user schedules a `khat` job that monopolizes a workstation. Conversely, a user could log in and prevent a scheduled `khat` job from running. Both these forms of denial of service can be detected. In fact, `khatd` could maintain a database of pending `khat` jobs, so that abuse of the system could be traced to the offending user.

3.5. Generating tickets for a user

When it is time for a batch job to run, the `khat` server must obtain a ticket for the user. It cannot simply issue a request for a ticket the way a user does because this request looks up the address from which it comes, and puts the client `IP` address into the ticket. Thus, the ticket must be constructed manually by the server. The steps taken by the server are as follows:

- Get the master key for Kerberos
- Get the TGT secret key from the Kerberos database
- Decrypt the TGT key with the master key
- Create a TGT ticket for the user
- Purge the master key and other keys from memory

A brief discussion of each of these steps follows. This discussion makes it clear that the `khat` server must have the Kerberos database available to it, and must either run on the same machine as Kerberos, or on a Kerberos slave machine.

3.5.1. Get the master key for Kerberos

The following steps are taken to get the master Kerberos key. We call `gettimeofday` to set the Kerberos time. Then we call routines to get and verify the master key. After that, the version number is checked. If the wrong version number appears, or if `kdb_verify_master_key` returns an error, we log the error and exit. In this case, no tickets can be generated, and mail is sent to the user. (The mail step is not yet implemented).

3.5.2. Get the TGT secret key from the Kerberos database

To get the TGT secret key from the Kerberos database, we call `check_princ`, which checks for expiration times on the master key and the service. It also populates the `Principal` data structure with information containing the key (encrypted under the master key), realm, *etc*.

3.5.3. Decrypt the TGT key with the master key

This step is straightforward. We use the master key to decrypt the TGT key.

3.5.4. Create a TGT ticket for the user

We have the TGT ticket that will be used to encrypt the ticket once it is constructed. Inside the ticket, we place user information such as name, instance, and realm, obtained from the call to `check_princ`. Then, we add the client host address that we obtained with a call to `getpeername` to see who the client was. In addition, we generate a random session key and include that in the ticket. Other usual information such as the time and ticket lifetime are also included. A lifetime of about 25.5 hours was chosen here, but that was arbitrary and based on the choice observed in existing Kerberos services. The ability to specify the lifetime could be added later as a user-specified option to `khat`.

3.5.5. Purge the master key and other keys from memory

Finally, we zero out all of the memory we used for highly sensitive information such as the master key. Even though this is a trusted machine, it never hurts to take added precautions.

3.5.6. Send the ticket to the client

Once the ticket has been constructed properly, it is encrypted under the session key available to the client and sent across the network. The client decrypts the ticket, and stores it with the batch job user as the owner, with no permissions for anyone else.

3.6. Renewing the tickets

If a batch job is to run for more than the lifetime of a TGT ticket, then the workstation must receive a new ticket for the user. The ability to renew tickets is a command line option. If a user prefers for jobs which outlive the tickets to die rather than have tickets generated until the job exits, she can chose to omit this option.

The `khat` program forks a master process to run the batch job. Before the job is actually run, this process forks a sub-process. This sub-process is in charge of maintaining the user's authentication on the workstation. When the master processing the job completes, it terminates the sub-process, and sends a message to the server that it is finished. This message is not necessary, but it is sent so that the `khatd` process on the server will exit normally. The details of this scheme follow.

The process that maintains the authentication works as follows. It sleeps until the authentication ticket is about to expire. Then, it calls `gettimeofday` to get the current time. The time is sent, encrypted, to the server, ensuring against replay, and proving possession of the secret key. The server checks the time, and if it matches to within a minute, is convinced that a new ticket must be sent. The server then constructs a new ticket for the user and sends it across the network. The workstation replaces its current ticket file with the new one, and then goes back to sleep.

The server on the other end waits for a request for tickets or a message from the client that the job has terminated. When a request comes in, the server checks that the time is within a minute of the current time. If not, the request is ignored, and no tickets are generated. If the time is correct, then the user creates a new TGT ticket for the user, encrypts it, sends it to the client, and continues waiting for requests.

4. KHAT runtime

A `khat` session consists of two phases. In phase one, the user invokes `khat`, and connects to a well known port on the server machine. When the spool file has been sent, the client and the server store away some information for future use and exit.

In phase two, it is time for a job to run, so the khat server connects to a well known port on the client machine, and the spool file and tickets are sent. This well known port is established by running a service, khatrun, on the client machine. Thus, the khat server becomes the khatrun client and the khat client becomes the khatrun server. For clarity, we will continue to use the terms client and server with respect to khat.

The server runs a program called khatcheck to determine when it is time to initiate phase two. Khatcheck runs in the background and wakes up every minute to see if there are any khat jobs to run. The name of the spool file contains the date and time for each job, so the current time is compared to the scheduled time for each job. If it is time, khatcheck forks and execs the khatrun program to begin phase two.

Khatrun needs some information for each job such as client name, instance, realm, and secret keys. Khatd, stores this information in a file before it exits. The name of this file contains the name of the spool file for the job as a substring so it can easily be associated with the correct job. This is possible because khatd is assumed to be running on a secure machine.

5. KHAT Utilities

We have written a couple of Kerberos services to give khat users some flexibility. The commands khatq and khatrm were added to display a list of pending jobs, to display the contents of a pending job, and to remove a pending job.

When these commands are invoked, the user's workstation receives a khat ticket, and mutual authentication is performed. The khat server is then presented with this ticket along with the name of the service being requested. At this point, the user and the khat server have established a secure communication channel, as they are both in possession of a secret session key. The client and server use this session key to encrypt all message traffic between them for the remainder of their communication.

5.1. The khatq command

When the khat server receives a request for the khatq service from the client, it creates a list of all pending jobs for that user. A string is then created containing the time and date for the job to run. In addition, a unique job number is assigned to each job based on a hash function of the user's uid and the time and date for the job. Currently, a four digit number is assigned, but to insure a higher probability of uniqueness, larger numbers can be used.

The list of pending jobs is sent to the client. Since the length of this list may vary, a terminator string is sent to the client at the end of the list. Since each khatq request will cause a new session key to be established between the client and server, and as the terminator string is only used once per session, this string is not distinguishable from other data once it is encrypted.

If there are no jobs scheduled for the user, then only the terminating string is sent. The client then prints a message that there are no jobs scheduled. If the user specifies a job number as an option to the khatq command, then the server will send the contents of the batch job, as originally submitted, to the client, which will then print them on the standard output. If the user specifies a job number that does not exist, or does not belong to that user, khatq prints an error message. To prevent users from discovering job numbers of other users, this error message does not reveal any substantive diagnostic information.

One weakness of the current implementation of khatq is that traffic analysis could reveal some information. By analyzing the messages between the client and the server, an eavesdropper can discover how many jobs are pending or the approximate length of a batch job. This can be solved by padding all the messages to a given length, adding redundant messages, or some combination of the two.

5.2. The khatrm command

When the khat server receives a request for the khatrm service, it verifies the ownership of the job, and then removes the spool file from the server machine. A message is sent to the client reporting success or failure. Again, a new session key is established for every invocation of khatrm, so this message cannot be distinguished from other data. The client either prints that the job was removed, or that it could not be removed. As in the khatq command, no further diagnostic information is printed so that it is impossible to discover if a job number represents a job scheduled by a different user.

6. Conclusions

Using khat, it is possible to run a batch job with authentication without manually renewing user tickets. The risks of having valid tickets on a vacant workstation are inherent to khat. To minimize the risk, the secret key used to authenticate to the server when the user is gone is maintained, encrypted in the process memory of the khat process on the client machine.

The khat server runs on a secure machine with access to the Kerberos database. It uses this database to generate tickets for users and the TGT service. Everything sent across the network is encrypted first with the secret key available on the workstation.

When scheduling jobs to run on a vacant workstation, there must be some security compromise for authentication to take place. This problem will remain until there is some way to reliably authenticate a workstation.

7. Future Work

We are considering some enhancements for future implementation.

At present, khat can run jobs only on the workstation from which they were scheduled. It would be convenient if the first available workstation or a user-specified workstation could be used. One possibility for a user-specified workstation to run khat jobs is to require that the user log into the target workstation, authenticate to khat, and then leave an encrypted session key around. However, this is no different from the user simply logging into the target workstation and scheduling the job from there.

Another possible extension is to add flexibility to the khat service. For example, a user may wish to specify that a workstation should maintain valid tickets only between certain hours, when presumably, she believes it is safer. An option can be added in which the user specifies the lifetime of tickets, and perhaps provides some conditions under which they should or should not be renewed.

Another feature which could be added to khat is the ability for a process to save its state before a ticket expires, send it (encrypted) to the server, and then wait until the user reauthenticates to continue running. This feature would be useful in a case where the workstation (somehow) realizes it has been compromised, or is about to be.

Availability

The code for khat can be obtained via anonymous FTP from /public/khat/khat.tar.Z at citi.umich.edu. AFS users can find the file khat.tar.Z in directory /afs/umich.edu/group/itd/citi/public/khat.

Complete instructions for building and installing khat can also be found in the same directory in the file khat.instructions. The system is meant for use with AFS or with Version 4 of Kerberos, and assumes the availability of the at command (in binary form).

Acknowledgements

The first author thanks David Snearline, Scott Dawson, Dan Muntz, and Michael Stolarchuk for technical pointers, and Bernie Galler for broad support.

We thank Steve Bellovin and Matt Bishop for their assistance.

References

1. R.M. Needham and M.D. Schroeder, ''Using Encryption for Authentication in Large Networks of Computers,'' *Communications of the ACM* **21**(12), pp. 993–999 (December, 1978).
2. J.G. Steiner, B.C. Neuman, and J.I. Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' pp. 191–202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).
3. B. Lampson, M. Abadi, M. Burrows, and E. Wobber, ''Authentication in Distributed Systems: Theory and Practice,'' *ACM Transactions on Computer Systems* **10**(4) (November, 1992).
4. National Bureau of Standards, ''Data Encryption Standard.,'' *Federal Information Processing Standards Publication*(46) (1977).

5. D. Kahn, *The Codebreakers,* Macmillan Publishing Co., New York (1967).

6. A. Kerckhoffs, *La Cryptographie Militaire,* Libraire Militaire de L. Baudoin & Cie., Paris (1883).

7. J.H. Saltzer and M.D. Schroeder, ''The Protection of Information in Computer Systems,'' *Proc. of the IEEE* **63**(9), pp. 1278–1307 (September, 1975).

8. V.L. Voydock and S.T. Kent, ''Security Mechanisms in High-Level Network Protocols,'' *Computing Surveys* **15**(2) (June, 1983).

9. G.W. Treese, ''Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD,'' *USENIX Winter Conference*, Dallas, Texas, pp. 175–182 (February, 1988).

The Architecture and Implementation of Network-Layer Security Under Unix

John Ioannidis
Columbia University
New York, NY 10027

ji@cs.columbia.edu

Matt Blaze
AT&T Bell Laboratories
Holmdel, NJ 07733

mab@research.att.com

ABSTRACT

swIPe is a network-layer security protocol for the IP protocol suite. This paper presents the architecture, design philosophy, and performance of an implementation of swIPe under several variants of Unix. swIPe provides authentication, integrity, and confidentiality of IP datagrams, and is completely compatible with the existing IP infrastructure. To maintain this compatibility, swIPe is implemented using an encapsulation protocol. Mechanism (the details of the protocol) is decoupled from policy (what and when to protect) and key management. swIPe under Unix is implemented using a virtual network interface. The parts of the implementation that process incoming and outgoing packets are entirely in the kernel; parameter setting and exception handling, however, are managed by user-level processes. The performance of swIPe on modern workstations is primarily limited only by the speed of the underlying authentication and encryption algorithms; the mechanism overhead is negligible in our prototype.

1. Introduction

Traditionally, system security has been addressed in an *ad-hoc* fashion and at a fairly high level, usually in the applications themselves. As applications become more distributed in nature and are relying more and more on the integrity of information received from their peers, it becomes attractive to consider common, general solutions to replace *ad-hoc*, application-specific security mechanisms. Most system-level security work has focused on the operating system and on arbitrating access to resources on a particular machine. Relatively little attention has been paid to securing communications in any consistent way. In modern distributed systems, however, which are characterized by a large number of single-user or single-purpose machines, the network itself is emerging as the best candidate for concentrating security efforts.

The explosively increasing size of the Internet, with its concomitant security problems, coupled with the wide range of services and applications that it supports, make IP-based networks a good choice for understanding and exploring network security issues. The rising mobility of users, the advent of wireless networking services, and the increasingly critical dependence of financial and commercial services on the Internet infrastructure, call for the adoption of authentication, integrity, and privacy features as *sine qua non* features of the network.

There have been a number of efforts aimed at providing security services at some layer in the network hierarchy [4,8,12,16]. However, no existing protocol is completely satisfactory for large-scale Internet-based deployment. Existing network-layer security protocols such as SP3 and NLSP implicitly impose a specific model of trust and policy; this manifests itself in the form of unwieldy features and a multiplicity of options. This unnecessary complexity frustrates efficient implementations and confuses mechanism with policy. Non-network-layer protocols, such as ADP and Kerberos, make it difficult to enforce security policies in a consistent manner, and often make it impossible to exploit their features without rewriting applications. Furthermore, non-network-layer solutions cannot be used to transparently enforce intermediate-hop security in an internetwork. swIPe's power lies in its minimalist structure; it avoids facilities that could best be provided elsewhere. Regardless, the techniques developed for swIPe can be readily adopted by implementations of any network-layer security protocol.

swIPe was designed to complement IP functionality by adding the necessary security features without altering the structure of IP itself. This is accomplished by encapsulating IP datagrams within IP datagrams of a new IP Protocol type (`IPPROTO_SWIPE`); these datagrams carry the payload of the original IP datagram, enough header information to reconstruct them at the remote end, and any additional security information that may be needed.

This approach has a number of advantages. First and foremost, since datagrams are encapsulated within other IP datagrams, they can transit parts of the network that know nothing about swIPe. Since the transport and higher-layer protocols never see swIPe datagrams, they need no changes and can employ the existing network infrastructure and interfaces. Since encapsulation and decapsulation can take place at any place where IP datagrams are processed (*i.e.*, either hosts or routers), swIPe may be used to implement both end-to-end and intermediate-hop security. Furthermore, a wide variety of security policies can be implemented simply by controlling when to encapsulate *outgoing* datagrams and when to accept *incoming* datagrams (which may or may not be swIPe-encapsulated). Finally, as our prototype experience demonstrates, swIPe can be implemented efficiently on a variety of modern Unix systems, such as SunOS, Mach, and BSDI.

swIPe provides three fundamental security services:

- Data confidentiality: protecting against unauthorized access to data being transmitted.
- Data integrity: protecting against alteration or future replay of traffic.
- Source authentication: network addresses are authenticated as part of the protocol.

It is important to point out that swIPe's protections are limited to the context of the network. Data not on the network itself, such as files, window contents, etc., may still require other protection mechanisms (*e.g.*, CFS [2]).

2. swIPe Protocol Overview

A complete specification of the swIPe protocol is beyond the scope of this paper. The reader is referred to [6] for the authoritative protocol description. This section gives a brief overview of the important features in swIPe.

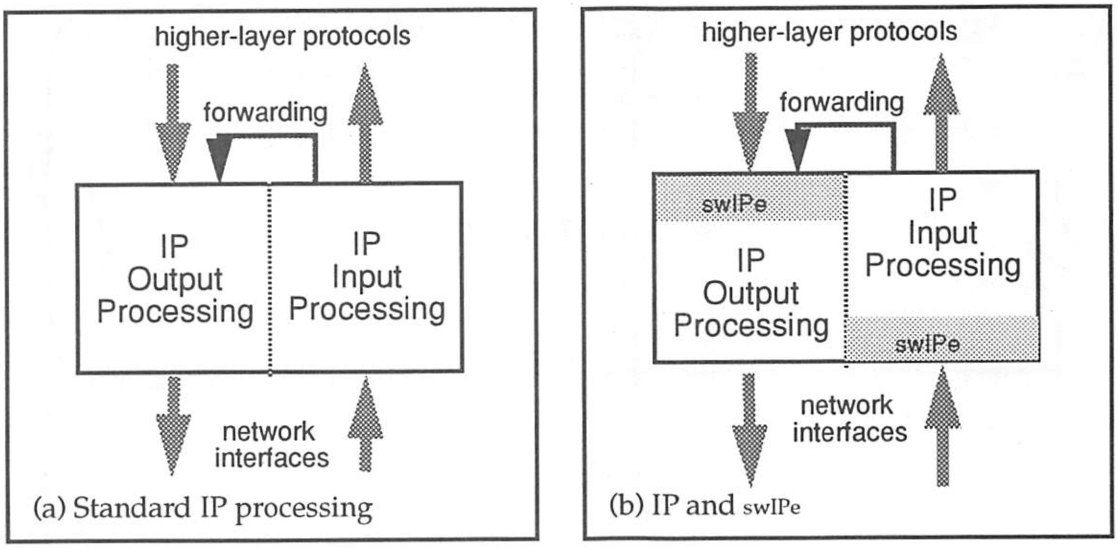
Briefly, a swIPe system consists of three conceptual entities on top of the ordinary IP mechanisms: the *security processing engine*, the *key management engine*, and the *policy engine*. This three-way split is entirely conceptual; an implementation need not maintain strict barriers between them.

The policy engine is responsible for three tasks: examining outgoing packets to determine whether they require swIPe processing, examining incoming packets to determine whether they are to be accepted, and deciding the exact nature of the processing. Note that the policy engine implements the local security policy for the host, which must be coordinated with the security of its correspondents.

The key management engine establishes the session cryptographic variables used by the security processing engine. It also communicates with key management engines on other hosts to establish key associations, and is responsible for managing any required secure key exchanges. Cryptographic variables can be transmitted with pre-arranged secret keys, with an authenticated Diffie-Hellman exchange, or through some other user-defined mechanism.

The security engine applies the actual authentication, integrity, and confidentiality processing on individual datagrams, as controlled by the policy engine, and using keys provided by the key management engine.

It is useful to think of swIPe as an input layer that processes datagrams between the data link layer and IP, and as an output layer that pre-processes datagrams just before they reach IP from higher layer protocols (see Figure 1). Actually, packets being routed – passed from the input side to the output side of IP –must also pass through the output swIPe layer.



(a) Standard IP processing

(b) IP and swIPe

Figure 1: IP Processing with and without swIPe

IP datagrams are encapsulated within swIPe packets using an extension of IPIP, the IP-inside-IP protocol (used by Mobile*IP, as defined in [5,7]). A swIPe packet is itself an IP datagram, containing an IP header with the IP protocol field set to 94. The payload of this packet consists of the swIPe header, which in turn contains control information to direct the receiving side on how to process the datagram, any necessary security information (such as an authentication value and sequence number (to protect against replay)), and the original IP datagram, possibly encrypted. The packet format is shown in Figure 2.

Conceptually, swIPe output processing proceeds as follows. The policy engine examines the IP packet (in the simple case, only its destination address), and determines whether the packet requires encryption, authentication, or both. If not, the packet is simply sent down to IP. Otherwise, a swIPe header is generated; the security processing engine obtains the keys from the key management engine, applies the appropriate authentication and encryption algorithms, and the resulting encapsulated packet is sent to IP for delivery. Note that it is possible that the key management engine may be unable to supply a key immediately (*e.g.*, a secure key exchange protocol may first have to be initiated with a remote host). In these cases, the implementation may either re-queue the packet until all keys are available, or simply drop it and rely on higher-layer protocols to retransmit it later.

Input processing proceeds in roughly the opposite manner. When an IP datagram is received, the swIPe policy engine examines it; if the packet is already a swIPe packet, it is passed to the security engine, which processes the packet in a manner analogous to the output processing described in the previous paragraph.

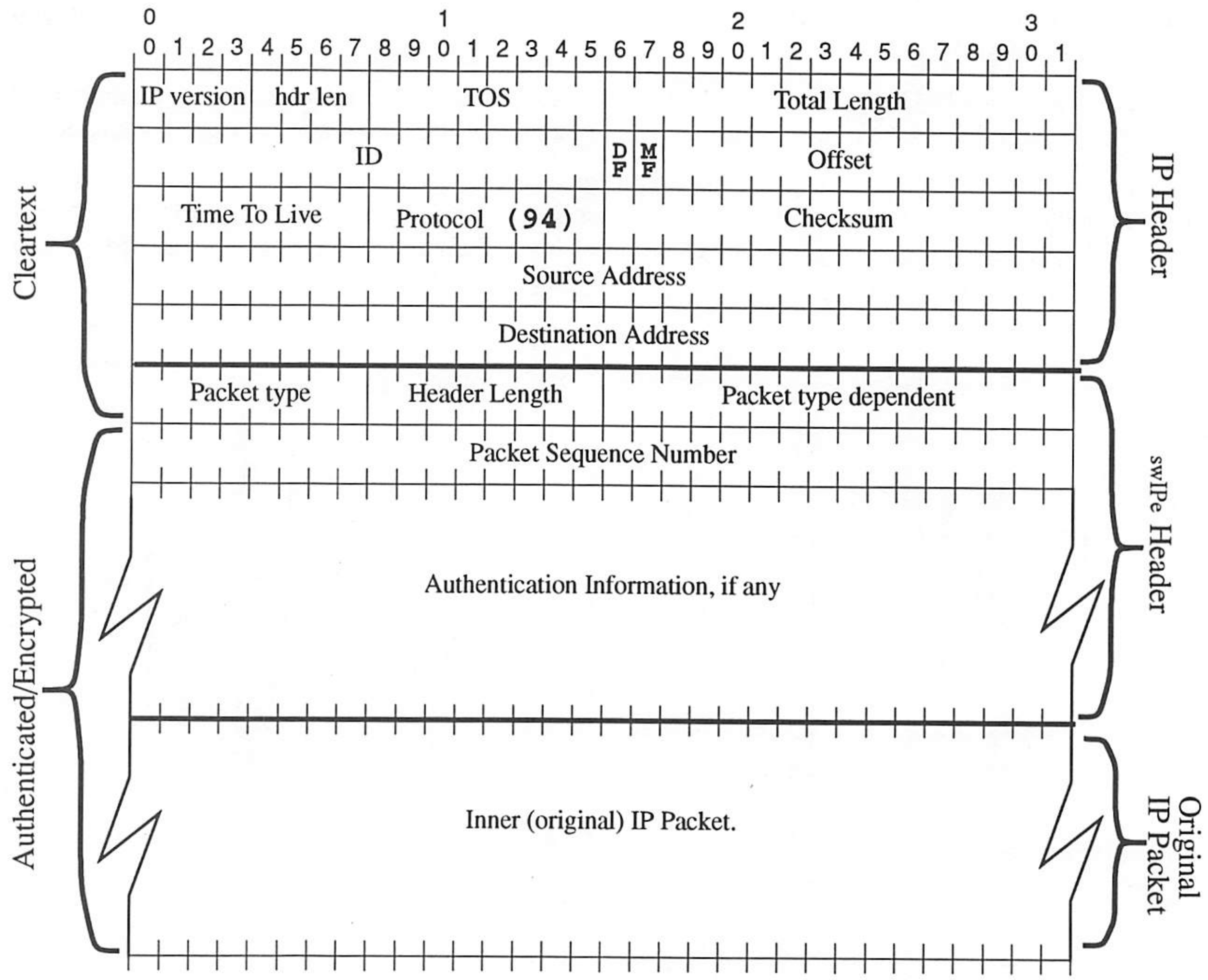


Figure 2: swIPe Packet Format

Otherwise, the policy engine determines whether the packet is admissible without authentication/encryption, and if so, simply passes it on to IP for regular input processing. If the packet is not admissible, it initiates exception processing, which could result in either a new key exchange, or simply an error message being recorded in the system log.

Note that the specifics of cryptographic variable exchange are not part of the swIPe protocol *per se*. swIPe provides an out-of-band mechanism for communication between key management agents through which necessary key exchanges take place.

Note also that swIPe does not require the use of any particular encryption or authentication scheme. Any encryption and cryptographic authentication functions can be used. Our implementation, described in the next section, supports DES [11] for encryption, and MD5 [14] seeded with a key for authentication. However, there is no reason why one could not use, *e.g.*, triple-DES, IDEA [9], or even Skipjack [10] for encryption, and SHS [13] for authentication, if desired.

swIPe can be used in a variety of configurations, each implementing a different security model. The simplest configuration is end-to-end security, as shown in Figure 3. Here, the individual hosts, `alice` and `bob`, use swIPe directly to communicate securely with each other. Both hosts must be equipped for swIPe processing.

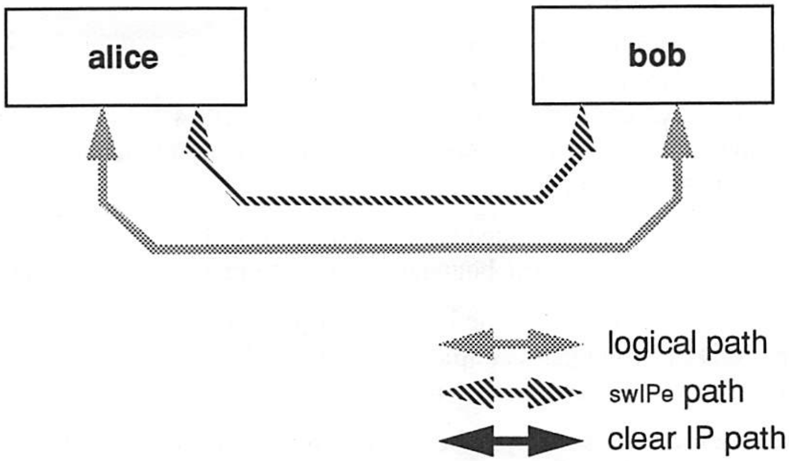
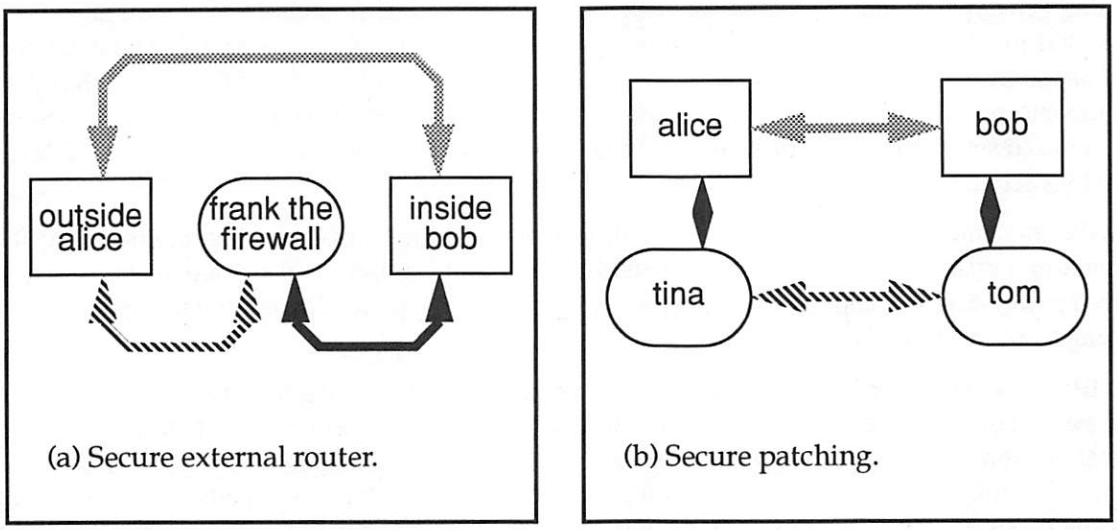


Figure 3: End-to-end swIPe operation

Other configurations are also possible in which the endpoints of communication are not swIPe-equipped. These configurations exploit the ability to run swIPe on routers as well as end hosts, and is the reason for encapsulating entire datagrams and not just the payload. Figure 4a shows the common network firewall as implemented under swIPe. The machine outside the firewall, `alice`, needs to communicate with the machine inside the firewall, `bob`. Since the firewall will not allow direct, unauthenticated IP connectivity, `alice` uses swIPe to deliver datagrams to `frank` (the firewall), which in turn decapsulates them and delivers them to `bob`. It is important to note that no machine inside the firewall (including `bob`) needs to run swIPe in order to securely communicate with outside hosts. This mode of operation can be extended to securely connect two or more networks over an untrusted link. Figure 4b shows `alice` and `bob` *tunneling* their traffic through a pair of swIPe firewalls, `tina` and `tom`.



(a) Secure external router.

(b) Secure patching.

Figure 4: Alternate swIPe configurations.

3. Implementation under 4.3 BSD.

In order to investigate the feasibility of network-layer security in practical environments and to provide a testbed for experimenting with various security polices, we built a prototype implementation of swIPe. Our prototype runs on several BSD-derived platforms, including SunOS 4.1.2 on Sun SparcStation-2s, and CMU's Mach 2.5 running on 486-class PCs. This section describes our implementation under BSD-Tahoe-derived Unix systems, in particular SunOS.

While our description of swIPe involved three conceptual entities (the security, policy, and key management engines), our implementation blurs the boundaries for performance and practicality reasons.

Part of the prototype is implemented within the Unix kernel (in particular, the security and policy engines, and the caching of session keys), and part is implemented at user level (in particular, most of the key management functions, as well as the policy configuration interface).

We have adopted a simple model of security policy, in which each host determines on a host-by-host basis the precise security transformations, acceptance policies, and keys to use. Future security policies should allow more sophisticated policy specifications, in which both coarser (groups of hosts) and finer (per-protocol or per-application) control is possible.

To simplify the implementation, we isolate all complex policy operations within easily modified user-level code. At the same time, performance requirements preclude consulting a user-level process for each packet. To balance these competing constraints, swIPe's operational parameters are specified through user-level interfaces to policy and key tables that are maintained in kernel space. Most packets can be processed entirely within the kernel simply by consulting those tables; only packets that cannot be processed according to existing table entries cause upcalls to user-level processing. swIPe adds four tables to the kernel. Two tables, the *input* and *output key tables*, contain the current session keys for a particular remote host; they are indexed by the address of the encapsulating packet. The session key entries also have a "time to live" which is decremented each time they are used. The other two kernel tables, the *policy tables*, describe how encapsulated and non-swIPe packets should be handled; they are indexed by the host IP address of the encapsulated (non-swIPe) packet. The input policy table describes the kind (if any) of swIPe processing (encryption, authentication, or both) that is expected of packets from a particular source IP address, and the address allowed to encapsulate packets for it. The output policy table contains the converse data, describing the kind of processing to perform for a given destination address and the address to which the encapsulated packets should be sent for it. Having two sets of tables, one for keys and one for policy, indexed separately, is the mechanism through which non-end-to-end configurations as described in the previous section are implemented.

The swIPe implementation includes kernel modifications for processing incoming and outgoing IP datagrams and processing policy table management `ioctl()` calls, a user-level *key management daemon*, a user-level *policy daemon*, and a set of tools for configuring swIPe policy (by issuing the appropriate table-management `ioctl()` calls).

The swIPe kernel code for processing outgoing datagrams is contained within a virtual network interface (called `sw`*n*, where *n* is `0`, `1`, etc.). This virtual interface appears to the rest of the system as if it were a conventional network interface; this technique is similar in spirit to the one described in [1]. Recall that conceptually, the policy engine filters all outgoing IP datagrams to determine whether the packet requires processing by swIPe. Since in our prototype only the destination IP address is used to select these packets, we use the existing BSD routing mechanism as our primary outgoing policy filter. At configuration time, a route is set to `sw0` for all hosts and networks with which swIPe processing is desired. It is also necessary to specify the kind of processing (encryption, authentication, or both) desired for each host and where the encapsulated packets should be sent. This is done through a special `ioctl()` call that creates an entry of the appropriate kind in the outgoing policy table. Another `ioctl()` sets up session key entries in the key table (with a time-to-live of `0`) for the encapsulating address. These `ioctl()`s are generally called by the key and policy daemons according to user-level configuration files.

When an outgoing packet is routed through `sw0`, swIPe first looks up the destination IP address in the output policy table to find the address to which the encapsulated packet should be sent. If no entry exists, the packet is sent, via the upcall mechanism (described below) to the policy management daemon. (Note that when no entry exists for an outgoing packet that was routed to sw0, it is because the destination host's address is on a network for which swIPe processing is required but to which no packets have previously been sent). If the lookup was successful, swIPe then looks up the encapsulating address in the output key table to find the current session keys for the appropriate operations. If this lookup fails, or if the keys are expired, the packet is sent via the upcall mechanism to the key management daemon. The packets that caused this exception processing are not delivered to their corresponding higher-layer protocol; their retransmission once a key is established is handled entirely by the higher-layer protocol.

Most outgoing packets have valid table entries and do not cause exceptions. These packets are encrypted and/or authenticated according to the policy rules and keys in the tables, and then encapsulated as the payload of a swIPe packet. The resulting packet is then sent to its destination (which may not be the same as the original packet) via the appropriate interface according to the standard routing tables (ignoring any `sw0` entries). The processing of outgoing packets in the kernel is summarized in Figure 5. Note that packets from the key management daemon bypass swIPe processing, since this traffic must usually complete as a prerequisite to passing any swIPe-protected data.

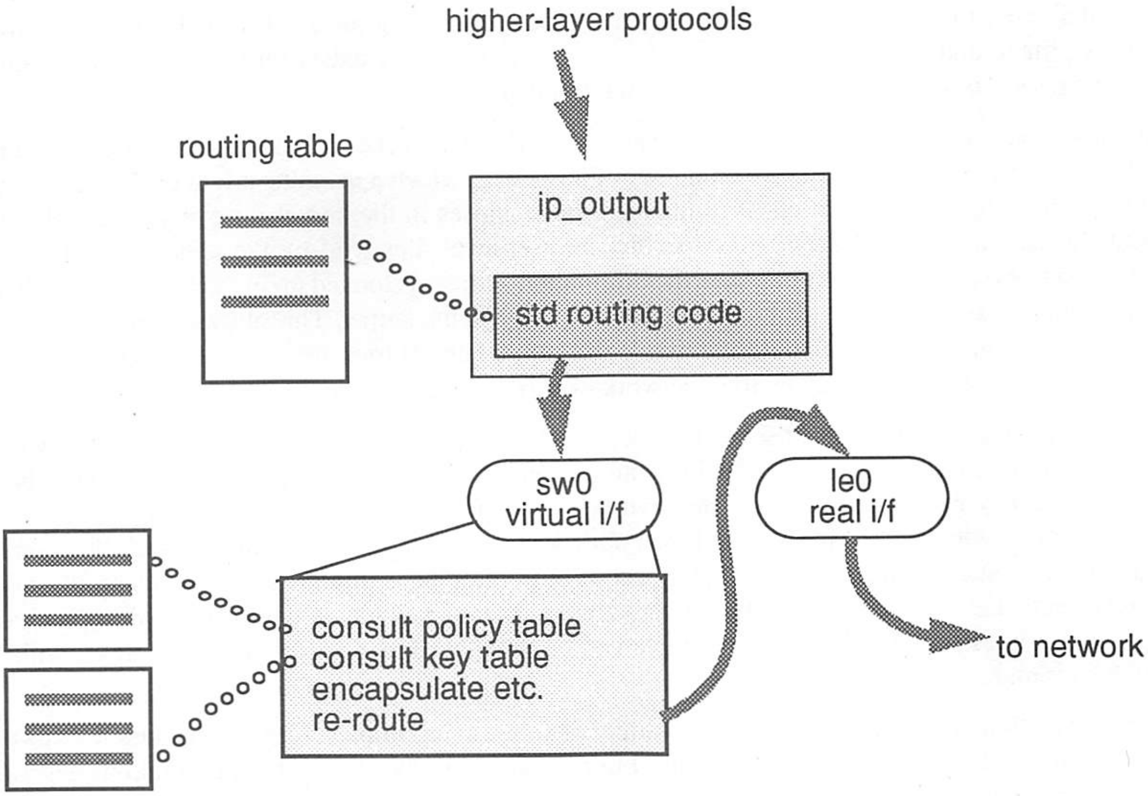


Figure 5: swIPe output processing.

Because no existing mechanism in 4.3BSD demultiplexes incoming packets by remote address (as the routing mechanism does for outgoing packets), input processing is more closely coupled to the rest of the kernel network code. swIPe modifies the standard kernel IP input routine to add a rudimentary packet filtering mechanism. The filter examines all incoming packets, whether they are swIPe-encapsulated or not, to determine if they should be accepted and passed on to the appropriate higher-layer protocol handler. The filter is controlled by the input policy table indexed by remote (source) IP host (or network) address, similar in structure to the output policy table.

Observe that there are several kinds of incoming packets that the filter must handle. First, consider non-swIPe packets that arrive. If they arrive from a host from which swIPe packets were expected (according to the table), they are dropped and a message is sent to the system log. Packets for which the table specifies that swIPe processing was not expected are passed on to the higher-layer protocol or routing mechanism. Unencapsulated packets that arrive from a host for which no table entry exists cause an exception and are sent to the policy daemon, which examines them and creates an appropriate table entry. Packets sent to the daemon are otherwise dropped. Note that packets intended for the key management engine (with protocol `IPPROTO_swIPe`) bypass these checks and are delivered directly.

Encapsulated swIPe packets are first decapsulated and decrypted/authenticated according to the keys in the input key table as indexed by the address of the encapsulating packet. If no key entry exists, or if the keys do not decrypt and authenticate properly, the packet is sent up to the key management daemon, which initiates a session key request with the remote host. Otherwise, the decapsulated packet's source address is looked up in the policy table to verify that its encapsulating packet came from the proper address and that the correct kind of processing (encryption, authentication) was applied. If everything is acceptable, the packet is delivered to the appropriate higher-layer protocol or routing mechanism. Otherwise, a console message is printed and the packet is dropped. If no input policy entry exists for the encapsulated source address, the packet is sent to the policy daemon and dropped.

Conceptually, the kernel policy and key tables can be thought of as cache entries for a database maintained by the policy and key daemons. Since the number of hosts with which a machine might exchange traffic is potentially very large, it is not practical to maintain table entries in the kernel for every potential correspondent. Instead, swIPe configuration is specified in user-level files read by the appropriate daemons. When a packet arrives at the policy daemon, the destination address is looked up in the appropriate file; if a match is found, an `ioctl()` is issued to configure the policy in the kernel. This allows policy to be specified concisely for large classes of addresses; while the kernel table entries are on a host-by-host basis, the daemon's tables can look up policy by their network or subnet address as well.

Key distribution uses either public or secret key techniques. In the public key version, each host's key management daemon maintains a list of RSA [15] public signature keys for each of the other swIPe hosts. When a session key is required, a Diffie-Hellman key exchange is initiated, with the components of the transaction signed with each host's private key. Public key distribution is static and manual; this is obviously an area for much future research. This aspect of our prototype is primarily useful as a proof of concept. In the secret key version, each host maintains a unique, static key for each other host which is used to encrypt the session keys. Again, the kernel table can be thought of as a cache of session keys, with the definitive keys managed by the daemon.

Our implementation of swIPe can be used in a variety of ways: in its simplest form, it provides host-to-host authentication and encryption. It can also be used to implement all the 'firewall' configurations described in the previous section.

The basic swIPe implementation adds less that 3000 lines of code to the kernel; the actual DES and MD5 code adds approximately another 3000 lines.

4. Performance

Our swIPe prototype is implemented entirely in software; this represents a somewhat unconventional approach to cryptographic protection of network traffic. Algorithms such as DES have traditionally been thought too slow to permit their use in high-bandwidth applications without adjunct hardware. However, key factors make software implementations of network encryption increasingly practical: modern workstations are becoming sufficiently powerful to execute cryptographic operations at speeds equal to previous-generation cryptographic processors; network bandwidth is not increasing at the same rate as processor speed, and finally, distributed applications rarely use even a small fraction of available network bandwidth. While our experience with swIPe reinforces our belief that encryption is becoming practical in software, swIPe itself does not depend on any particular software or hardware architecture.

We instrumented the SunOS 4.1.2 kernel to measure the costs of the various components of our implementation. All measurements were taken by using the internal SparcStation microsecond-resolution clock. The clock was read in key locations of the code being timed, and the resulting timestamps and time intervals were deposited in a kernel buffer. A user-level process would periodically empty the buffer and record the results. The measurement error was within approximately 4 microseconds.

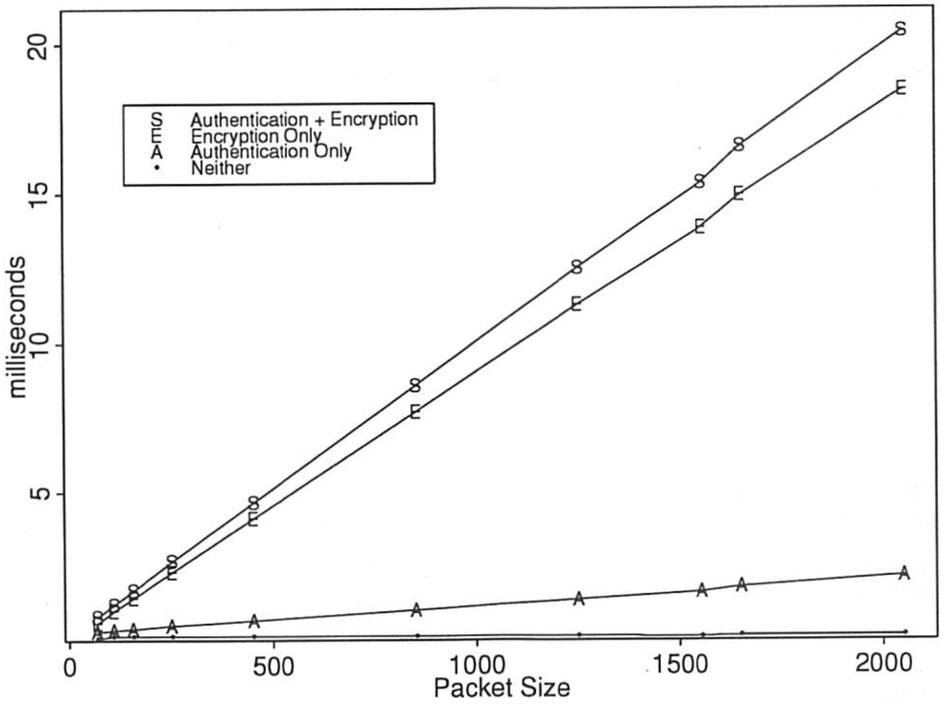


Figure 6: swIPe processing overhead

The overhead of the swIPe protocol processing itself is minimal, adding approximately 100 microseconds to the total packet processing time regardless of packet size. For small datagrams, this is about half the time it takes the kernel to deliver them to the hardware interface, diminishing further in significance as the datagram size increases. This overhead is dwarfed by the amount of time it takes to compute a cryptographic hash function, which in turn is dwarfed by the time it takes to perform a single pass DES in software. On a SparcStation-2 class machine, processing a 100-octet datagram adds roughly 200 microseconds for computing an MD5 cryptographic checksum, and another 900 microseconds for encrypting using a highly-optimized software implementation of DES. Processing a 1000-octet packet requires roughly 1000 microseconds for the MD5 checksum and 10,000 microseconds for the DES encryption. Figure 6 shows the total time required to process a swIPe packet in the kernel as a function of the packet size. The four curves show the processing time the four possible processing combinations: no protection, only authentication (MD5), only encryption (DES), and both authentication and encryption.

The timing ICMP ECHO packets is consistent with these numbers. Namely, a 56-octet ping gives a round-trip delay of 8 milliseconds, while a 1000-octet ping gives a round-trip delay of 47milliseconds. These numbers include four encryption/decryptions and four authentication operations (when the packet is sent, received, returned, and received again). TCP throughput between two SparcStation-2s on a lightly loaded Ethernet is approximately 440kbps when using DES encryption; the observed variance (~30kbps) obliterates any fine-grained measurement of the added effect of authentication. When using MD5 authentication only, we observed TCP throughput at approximately 3400kbps. For comparison, the TCP throughput of the unmodified machines (without swIPe) is roughly 6000kbps.

It is important to put these numbers in perspective. Interactive traffic consists mostly of small packets; hence, adding authentication and encryption between communicating hosts adds a fixed delay of 2-3 milliseconds, which is not perceptible considering that this is usually the delay associated with going through a single router. Large packets are usually associated with batch transfers or remote filesystem operations, and are usually already bound by disk I/O rates rather than network rates. Note that on a SparcStation-2, swIPe can process datagrams at roughly one megabit per second, which, although not transparent, is at least within an order of magnitude of the bandwidth of a typical NFS server. Of course, hardware-assisted cryptographic functions can further reduce the effects of this bottleneck, as could the use of lower latency cryptographic algorithms (e.g., DES's OFB mode, which can use idle processor cycles to precompute cryptographic masks).

5. Conclusions

In developing a network security protocol, it was important to keep both interoperability with existing systems and a migration path in mind. To this end, swIPe functionality does not depend on any IP-specific features, and can therefore be easily adapted to other connectionless network protocols.

swIPe is not only a fully-functional network security protocol in itself, but also provides the framework for experimenting with alternative security algorithms and policies, and, more important, large-scale key management strategies. It is perhaps in these areas that we will see much of the interesting research in network security systems. It is our hope that researchers and developers of network and other large-scale security systems will adopt swIPe as a vehicle; while the current implementation is by no means stable enough for general distribution, interested parties are encouraged to discuss their plans with the authors.

Acknowledgments

This work owes its conception to many fruitful discussions, guidance, and encouragement by Phil Karn. The implementation of the encryption code was aided greatly by discussions with Jack Lacy. The implementation and testing were carried out over many late evenings at the Mobile Computing Lab at Columbia University.

References

[1] S. M. Bellovin. Pseudo-Network Drivers and Virtual Networks – Extended Abstract. In *Usenix Conference Proceedings*, pages 229–244. Usenix, January 1990.

[2] Matt Blaze. A cryptographic file system for Unix. In *First ACM Conference on Communications and Computing Security*, Fairfax, VA, November 1993.

[3] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*}, IT-22:644–654, 1976.

[4] D. P. Anderson *et al.* A Protocol for Secure Communication in Large Distributed Systems. Technical Report UCB/CSD 87/342, University of California, Berkeley, February 1987.

[5] John Ioannidis. *Protocols for Mobile Internetworking*. Ph.D. thesis, Columbia University in the City of New York, 1993.

[6] John Ioannidis, Matt Blaze, and Phil Karn. swIPe: The IP Security Protocol. *To appear.*

[7] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. IP-Based Protocols for Mobile Internetworking. In *Proceedings of SIGCOMM'91*, pages 235–245. ACM, September 1991.

[8] ISO/IEC JTC1/SC6. ISO-IEC DIS 11577 – Information Technology –Telecommunications and Information Exchange Between Systems – Network Layer Security Protocol, November 29 1992.

[9] X. Lai and J. Massey. A Proposal for a New Block Encryption Standard. *Proc. EUROCRYPT 90*, pages 389–404, 1990.

[10]Mykotronx, Inc. MYK-78 Encryption/Decryption Device. Data Sheet, 1993.

[11]National Bureau of Standards. *FIPS Publication 46-1: Data Encryption Standard*, January 1988.

[12]National Institute of Standards and Technology. *NISTIR 90-4250: Secure Data Network System (SDNS) Network, Transport, and Message Security Protocols*, February 1990.

[13]National Institute of Standards and Technology. *Publication YY: Announcement and Specifications for a Secure Hash Standard (SHS)*, January 22, 1992.

[14]R.L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Internet Activities Board, April 1992.

[15]R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[16]J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. *Proc. Usenix Winter Conference*, February 1988.

Retrofitting network security to third-party applications — the SecureBase experience

Jonathan I. Kamens
OpenVision Technologies, Inc.
jik@security.ov.com

Abstract

Systems such as Kerberos, designed to provide secure user and service authentication over insecure open networks, continue to gain acceptance in the UNIX world. There are both freely available and commercial products which reduce the vulnerabilities inherent in trusting "traditional" UNIX security in a distributed environment. However, such products generally do not provide similar protection for third-party applications which have their own network security paradigms. This paper shows how such protection can be extended to encompass SYBASE, a widely-used, distributed database system. The experience with SYBASE is generalized to other network applications.

1 Introduction

As distributed computing has become more prevalent, computer security technology has evolved to meet the demand for robust security against network-based attacks. However, many third-party applications have failed to keep up with this evolution and show no signs of preparing to catch up in the near future; these applications are therefore often unacceptably vulnerable to network attacks.

Since it is often impossible for the end-user site to make source-code-level changes to third-party applications, and since waiting for vendors to fix major security problems can often lead to a very long wait indeed, some other method must be found for improving security. Although not every third-party application can be successfully secured, and no universal solution exists for securing all applications, some classes of applications are particularly well-suited to external security enhancements. Relational database technology is one such class.

In this paper, we describe the approach we used to develop SecureBase, a product which enhances the security of a specific relational database system, SYBASE,[1] in a way that does not require modifications to any SYBASE servers or clients. The limitations of our approach are also discussed. After illustrating the feasibility of our approach with this specific example, we discuss in more general terms how it can be applied to other applications.

2 Overview of our approach

We were asked by a client to analyze the network security present in Sybase's client-server products and to develop solutions for its problems. The approach we used consists of these steps:

[1] The system is SYBASE and the vendor is Sybase; i.e., you purchase SYBASE from Sybase.

- Determine what security problems exist in the current implementation.
- Determine what techniques can be applied to solving the problems.
- Rank the importance of the security problems and of the advantages and disadvantages of each technique.
- Choose one or more techniques to use in the solution.
- Design and implement the solution using the chosen technique(s).

In Sections 3 through 7, we discuss the application of this approach to the SYBASE problem, and in Section 8, we discuss the impact of our solution on performance. In Section 9, we use our experience with SYBASE as the basis of a more detailed overview of our approach.

3 SYBASE's authentication technology – the *status quo*

3.1 Standard SYBASE network access

SYBASE clients access servers over the network using Sybase's Open Server protocol. The client side of this protocol is accessible to application developers in the SYBASE DB-Library, an API for establishing connections to servers, transmitting queries, and receiving responses. The server side of the protocol is likewise accessible in the SYBASE Server-Library API, which allows developers to implement Open Servers which accept connections from clients, receive queries, and transmit responses. In this paper, *SQL server* refers to a SYBASE SQL Server; *application server* refers to a server that is not a SQL server but uses the Server-Library API; and *Open Server* refers to any server that understands the Open Server protocol, i.e., either a SQL server or an application server. No distinction is made in the DB-Library between establishing a connection to an application server and establishing a connection to a SQL server. An Open Server can itself be a DB-Library client, i.e., a server can act as a client of another server.

A client connects to a server using the DB-Library's `dbopen` call. Included in the information transmitted from the client to the server during the `dbopen` is information which authenticates the client to the server: the username requesting the connection, and the associated password. In addition, the client can include *remote passwords,* consisting of a list of { *server name, password* } couplets, to be used by the server if it needs to establish a connection to another server on the client's behalf.[2]

When a SQL server receives an incoming connection request, it looks up the specified username in a SQL table of database users and verifies that the password given by the client matches the password in the table. If it does, then the connection is allowed; if not, an error is returned and the connection is terminated.

Since no distinction is made between connections to application servers and connections to SQL servers, application servers receive the same authentication information (username, password, remote passwords) as SQL servers. However, their use of the data is application-defined. In other words, password verification is not handled automatically by the Server-Library.

Typed data is transferred between clients and servers using Sybase's Tabular Data Stream (TDS) protocol, which deals with byte-order differences, differences in floating point representations, etc., for the application.[3] Although the TDS protocol does not affect Sybase's authentication directly, it becomes relevant when considering how to improve the authentication, as discussed in Section 4.

[2] Unfortunately, the remote password functionality is limited by the fact that the username to use on the remote server cannot be specified — it must be the same as the primary username in the `dbopen` call.

[3] In other words, the TDS protocol is similar in function to Sun RPC's XDR standard [3].

3.2 Security problems with the Open Server protocol

The Open Server protocol is vulnerable to four different attacks:

Password theft. Client usernames, passwords and remote passwords are transmitted over the network in cleartext (i.e., without encryption). As a result, anyone who can read all packets on the network between a client and a server ("snoop" on the network) can steal usernames and passwords and use them later to access servers illicitly. Such passive monitoring is very difficult to detect, and stolen passwords can remain a threat long after monitoring has ceased (since users often use the same passwords for long periods of time).

Data theft. Data sent between clients and servers can also be stolen, since the Open Server protocol does not provide any form of data encryption. This attack is completely passive (as opposed to password theft, which requires the active step of *using* stolen passwords before the attacker accrues any benefit) and therefore nearly impossible to detect.

Unauthorized password use. The Open Server protocol makes the assumption that servers can be trusted. However, an untrustworthy server (or the administrator of an untrustworthy server, or an attacker who has broken into a server) can use a client's remote passwords (or the password used to authenticate to the server, since users tend to use the same passwords for different servers) to gain illicit access to other servers.

Data-stream modification. There is no provision in the protocol for verifying data integrity. A clever attacker can insert data into the client-server data stream and (depending on the network technology and topology) may also be able to modify the data stream by deleting data from it and/or changing data in it. As network protocol cracking tools become more widely distributed, this attack becomes accessible to a wider range of attackers.

3.3 Administrative problems with SYBASE security

A SQL server keeps its own table of usernames and passwords, independent of, and administered separately from, any enterprise-wide authentication database, e.g., a NIS passwd map or a Kerberos [2] database. This complicates things for administrators, who would like to be able to easily change user passwords or revoke access, and for users, who would like to use only one username and password (each) for all of their authentication needs. Furthermore, when users are forced to remember multiple passwords, they tend to write them down, which significantly decreases the security of the system.

User passwords are stored on a SQL server in cleartext (as opposed to UNIX passwords, for example, which are stored in encrypted form). Therefore, any attacker managing to break into the SQL server with system administrator access immediately has access to all passwords and can subsequently use them for illicit access to the server (and possibly to others).

Finally, the Server-Library provides no built-in functionality for storing and verifying passwords, so application servers designed by an organization will have to keep independent authentication databases.

3.4 A specific example: stealing SYBASE passwords

When a DB-Library client initiates a connection to an Open Server, it sends a packet to the server with all of its login information in it (including the authentication information discussed above). The format of this packet is both uniform and predictable. Therefore, an attacker can easily determine how to detect such packets and extract from them usernames and passwords for a server. Furthermore, if the attacker waits long enough, it is likely that someone will log into the server over the network using the system administrator username and password, and once the attacker has them, he has unlimited access to the server.

This attack requires the ability to snoop on the network over which packets between the client and the server are transmitted. Ready-made tools for snooping are quite common and becoming more so — if an attacker with network access wants to snoop, he probably can. Furthermore, even if all authorized hosts on a network are secure and therefore can't be used for snooping, an attacker can probably still snoop, e.g., by unplugging a workstation from the network and replacing it with a portable computer (with his ready-made snooping tools installed).

For example, in a recent demo, we were able to steal SQL Server usernames and passwords using a SPARCbook, the SunOS `etherfind` command, and a `Perl` script with only 21 lines of code.

4 Potential solutions to SYBASE's network-security problems

The potential solutions we considered comprise three general approaches: network security, connection encryption and connection authentication. Specific solutions using each of these approaches are discussed below.

4.1 Securing the entire network

One technique is to eliminate the possibility of snooping on the network. There are two ways to do this:

- Make the network physically secure, so that an attacker cannot plug his own snooping hardware into it (or install a tap to see what's going over the network), and make all the machines on the network secure, so that an attacker cannot snoop from one of them, or

- Protect all data on the network with link encryption.

If the network is secured, then password theft, data theft and data modification are prevented.

Physically securing a network may be practical for a small network in a uniform, tightly administered environment. However, as the network gets larger and more distributed, it becomes much more difficult to establish a single (physical) network security policy, and even more difficult (financially and administratively) to enforce it. Furthermore, as the complexity of the network increases, it becomes more and more likely that a mistake will be made somewhere and that overall network security will be compromised as a result. Finally, physically securing the network is not an option if a portion of it is not controlled by the organization, e.g., if two offices in different locations are using one of the commercial Internet backbones for traffic between them.

Although products exist for encrypting all traffic on a network, such encryption is also impractical on a large network, for a number of reasons:

- Network encryption offerings tend to be expensive (in dollars, as opposed to MIPS or network bandwidth, although they are often expensive in those areas as well), and as the size of the network increases, so does the cost of encryption (because encryption hardware and/or software is usually required for each machine on the network).

- Network encryption brings with it a significant administrative overhead. The encryption technology itself must be managed, and furthermore, the encryption can make it more difficult to manage the network (i.e., it may prevent the licit network snooping that is sometimes necessary in order to diagnose network problems).

- A site with a heterogeneous network environment (in terms of the networking technology used, and in terms of the kinds of machines used) may find that no network encryption offering will support the entire environment.

- Network encryption of all network traffic when only some of the traffic needs to be protected is overkill, especially when the traffic level on the network is high enough that the encryption causes noticeable performance degradation.

- As with physical network security, network encryption of traffic over a commercial Internet backbone may not be possible.

4.2 Connection encryption

If securing or encrypting the entire network is not feasible, another technique is to secure individual Open Server connections from snooping, by encrypting individual connections in software. If done properly, this eliminates password theft, data theft and data modification, since passwords and data can be encrypted and signed to preserve their secrecy and integrity. The following sections discuss in more detail how connection encryption might be implemented.

4.2.1 Application encryption

An organization can implement DB-Library clients and application servers which encrypt/decrypt transmitted/received data. An advantage of handling encryption in the applications is that they can selectively choose which data needs to be encrypted; this may be important if a significant portion of the data being transferred is not secret.

This technique has some problems:

- Without access to the communication channel between the client and server, encryption must be done above the TDS protocol layer. As a result, all of the automatic type-conversion functionality of the Open Server protocol is unusable.

  Current Sybase development libraries do not support access to the communications channel of the Open Server protocol below the TDS level,[4] so it seems unlikely that a TDS-capable encrypted Open Server connection can be implemented without extensive help from Sybase.

- Even if it is possible to implement DB-Library clients and application servers which encrypt and decrypt data below the TDS layer, it is not possible to modify SQL servers to do the same,[5] so encrypted communication with SQL servers is not possible.

- Many organizations purchase ready-made DB-Library clients and application servers which cannot be recompiled or relinked. It is impossible to engage in encrypted communication with these, just as it is impossible to do so with SQL servers.

4.2.2 Socket-level encryption with mediators

Treese and Wolman [5] discuss rerouting client-server network connections through *application relays* to enhance security. One example they give is an application relay which accepts incoming X Window System

[4] Sybase plans to address this problem in a release some time in 1994.

[5] SQL servers are sold in binary-only form by Sybase and the cost of obtaining their source code is prohibitive.

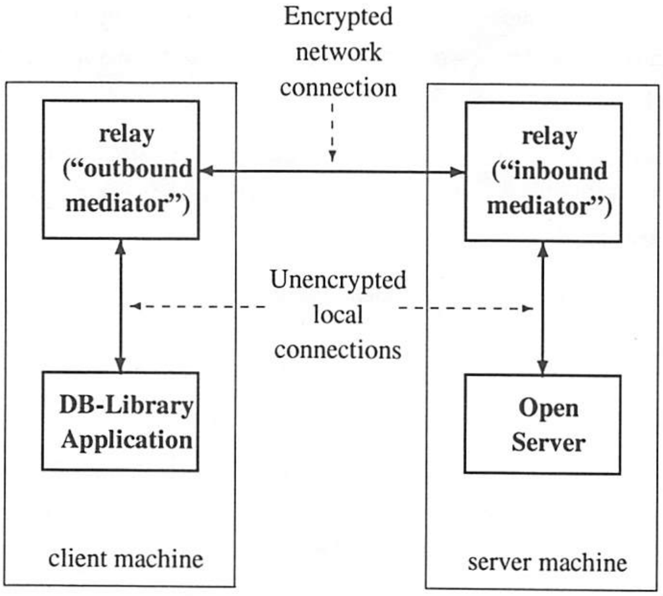


Figure 1: Connection encryption with mediators

connections for a user and asks the user if each connection should be permitted. Permitted connections are forwarded to the user's actual X server, using TCP protocol forwarding.

A similar approach can be taken to encrypt a SYBASE client-server connection. A DB-Library application can be fooled into connecting to an application relay running on the client's local machine, rather than to the remote server.[6] Since the relay is running on the same machine as the DB-Library application, communication between the two is not transmitted over the network and is therefore not vulnerable to snooping.[7] This relay then connects to another relay on the remote machine of the desired Open Server. The remote relay then connects to the Open Server (and again, communication between the two is local to the machine and therefore not transmitted over the network). The two relays pass all data between the client and server, but encrypt it before network transmission (and decrypt it after reception). This system is illustrated in Figure 1.

There are important differences between this system and Treese and Wolman's application relays. First, two relays are required, one on the client machine and one on the server machine. Second, the primary purpose of Treese and Wolman's system is to allow limited, policy-restricted connections through a firewall, whereas the purpose of the encrypting relay system is to provide a protected communication path between a client and a server.[8] To distinguish between Treese and Wolman's one-relay approach and the two-relay approach presented here, we refer to our relays as *mediators*. A mediator which accepts a connection from a client and forwards it through an encrypted connection to another mediator is called an *outbound mediator*, and a mediator which does the opposite, accepting an encrypted connection from another mediator and forwarding it to a server, is called an *inbound mediator*.

[6] This is done by modifying the `interfaces` file, which is used by Open Server clients to locate server hosts and port numbers, so that entries in it point to a server running on the local machine rather than to real servers on other hosts.

[7] This assumes that the underlying operating system short-circuits network connections to the local host so that their data is not transmitted on the network (SunOS 4.1.3 meets this condition). If the OS doesn't short-circuit local connections, it can usually be fooled into doing so with routing table changes. Unfortunately, the intuitive method of using the loopback address (`127.0.0.1`) can't be used to force a short-circuited connection, at least not when dealing with SYBASE in particular, because there is a bug in SYBASE which prevents the loopback address from working in `interfaces` files.

[8] The system proposed here can be made to work through a firewall by placing a third relay on the firewall machine in the middle of the encrypted communication channel.

This technique has none of the problems of application encryption. Furthermore, it is very generalized; a mediator implementation to encrypt Open Server connections will work with only slight modifications with other types of TCP connections as well.

However, it, too, has its problems:

- Open Server connections aren't always carried over TCP/IP networks with machines at both ends that can be made to do TCP protocol forwarding easily. Different mediators have to be implemented for each network type supported. Application encryption, on the other hand, is implemented on top of the Open Server protocol, which means that it should be easily portable to different network types.
- There may be significant performance degradation caused by forcing all client-server communication to pass through two extra processes.
- Since encryption takes place below the application level, all data must be encrypted, even data that does not have to be kept secret. This is a problem if encryption of large amounts of non-secret data causes performance degradation.

4.2.3 Open-Server-level encryption with mediators

If, instead of implementing mediators at the TCP level as described above, they are implemented on top of the Open Server libraries,[9] then they will be portable to all network types that the libraries support.

However, this increased portability is obtained at the expense of bringing back one of the major problems of application encryption. Until the Open Server libraries support access to the communication stream, proper encryption of data by an Open-Server-level mediator can be done only with extensive help from Sybase.

4.3 Connection authentication

Rather than relying on the Open Server username/password mechanism to authenticate users, a stronger authentication system such as Kerberos or SPX [4] can be integrated into the system.

If the authentication system chosen does not require private data to be sent by a client to a service during authentication and if it obviates the need for separate authentication databases in Open Servers (both Kerberos and SPX meet these conditions), then password theft and unauthorized password use attacks are prevented.

Just as with connection encryption, authentication can take place at three levels: application, socket and Open Server.

4.3.1 Application-level authentication

Strong authentication can be layered on top of the DB-Library and Server-Library, and DB-Library clients and Server-Library servers can be modified to use the new authentication. However, just as with application encryption, this technique will not work with SQL servers or with third-party clients and servers that cannot be relinked or recompiled.

4.3.2 Socket-level mediator authentication

If a mediator can be made to understand only enough of the Open Server protocol to be able to distinguish the username and password sent at the beginning of an Open Server data stream, then mediators can be used to do

[9] The Open Server libraries provide specific functionality, "TDS pass-through mode," that a server can use to transparently pass data between a client and another server.

Table 1: Advantages of techniques for securing SYBASE

	Securing the network	Encryption			Authentication		
		Appli-cation	Socket	Open Server	Appli-cation	Socket	Open Server
Prevents password theft	√	√	√	√	√	√	√
Prevents unauthorized password use					√	√	√
Prevents data theft	√	√	√	√			
Prevents data modification	√	√	√	√			
Eliminates Open Server authentication databases					√	√	√
Works with all networks supported by Open Server	√	√		√	√		√
Easily supports other application protocols	√		√			√	
Easily supports multiple Open Server protocol versions	√	√		√	√		√
Works with preexisting clients and servers	√		√	√		√	√
Supports full TDS functionality of Open Server libraries	√		√		√	√	√
Causes negligible performance degradation					√		

strong authentication. This has some of the same disadvantages as socket-level mediator encryption: different mediators have to be implemented for the different network types supported by the Open Server libraries, and performance may suffer because client-server communication must pass through two extra processes.

Furthermore, since the Open Server libraries support multiple versions of the Open Server protocol, the mediators will have to know how to recognize different versions of the protocol and change their technique for locating the username and password appropriately. Also, the mediators may have to be updated when new protocol versions are released.

4.3.3 Open-Server-level mediator authentication

Mediators can be implemented using the Open Server libraries to receive incoming connections, initiate outgoing connections, perform necessary authentication using Open Server remote procedure calls, and then enter pass-through mode so that the end client and server can communicate with each other. This technique does not suffer from most of the problems of application authentication and socket-level mediator authentication; however, the performance impact of forcing client-server communication to go through extra processes is still a concern.

4.4 Summary

Table 1 summarizes the advantages of the various techniques discussed above.

Table 2: Prioritization of solution advantages

	Required	Desirable
Prevents password theft	√	
Prevents unauthorized password use		√
Prevents data theft		√
Prevents data modification		√
Eliminates Open Server authentication databases		√
Works with all networks supported by Open Server	√	
Easily supports other application protocols		√
Easily supports multiple Open Server protocol versions		√
Works with preexisting clients and servers	√	
Supports full TDS functionality of Open Server libraries	√	
Causes negligible performance degradation		√

5 Ranking problems, advantages and disadvantages

We chose a simple system for ranking the problems, advantages and disadvantages listed in Table 1. By negating the disadvantages and treating them as advantages, we were able to do all of our ranking on one scale. We chose simply to classify each item in the table as either required or desirable. Our ranking is shown in Table 2.

Our prioritization was, for the most part, client-driven, since we were asked to design a solution to SYBASE's problems by a client with specific requirements. As shown in the ranking table, their primary security concern was theft of passwords by network snoopers, and their other concerns were related to usability in their environment and interference with Open Server library functionality.

6 Choosing techniques

Comparing the requirements in Table 2 with each of the techniques in Table 1, we saw that two of techniques, securing the network and Open-Server-level mediator authentication, satisfy all the requirements. We and our client both rejected securing the network as a valid solution, for the reasons given in Section 4.1. Therefore, we chose Open-Server-level mediator authentication, using Kerberos version 4 as our strong authentication system, as the basis for SecureBase.

6.1 Why we chose Kerberos

We chose Kerberos version 4 as the basis for our solution for two primary reasons:

- Stable, reliable implementations of Kerberos version 4 exist, both in freely redistributable and commercial forms. We can bundle a free implementation with our product for sites that are not already running Kerberos, or refer them to a commercial vendor, or use an existing Kerberos installation.
- Kerberos is currently more of a standard, both *de facto* (through widespread use) and *de jure* (through standards-body and vendor acceptance), than any of the other available strong authentication technologies. It is not likely to lose this acceptance in the near future.

Although most vendors and standards bodies endorse Kerberos version 5 instead of version 4, primarily through the Open Software Foundation Distributed Computing Environment [1], we felt that version 5 offerings are not yet stable enough or available widely enough to base a short-term product (which SecureBase was intended to be) on it. In any case, migration of a product and its users from Kerberos version 4 to Kerberos version 5 is likely to be less arduous than migration from any other strong authentication technology to Kerberos version 5.

6.2 Enhancements to our primary solution

In addition to meeting our requirements, our chosen technique also met several of our desirables. However, several desirables were still missing. We dealt with them as follows:

Data-theft and -modification prevention. We are working with Sybase in an effort to find out enough about the Open Server protocol that we will be able to encrypt a TDS pass-through mode communication stream in our mediators. We are also considering adding socket-level mediator encryption underneath our Open-Server-level mediator authentication. We have decided to postpone work in this area until the next version of SecureBase.

Easy support for other application protocols. Although our choice to implement Open-Server-level mediators rather than Socket-level mediators means more work for us if we want to add authentication to a system besides SYBASE, it also guarantees easy portability to all supported Open Server networks and platforms as well as automatic support for multiple Open Server versions. We felt that these advantages, which greatly impact the usability of the product, were compelling enough to force us to choose Open-Server-level mediation.

Negligible performance degradation. The use of mediators is unavoidable when adding strong authentication to existing applications. However, applications that can be relinked or recompiled can be modified to do strong authentication directly, at the application level. We therefore decided to provide application authentication functionality in addition to our mediator authentication functionality. In other words, we would provide an API for developers to use to do the same authentication that the mediators do, but directly, rather than through mediators. As a result, performance degradation can be reduced for applications that can be relinked or recompiled.

7 SecureBase's design

Our design consists of several components:

- An API library for use by DB-Library clients to establish Kerberos-authenticated connections to Kerberos-capable Open Servers, including inbound mediators on remote hosts. A client using this API is a *SecureBase client.*

- An API library for use by Server-Library servers (*SecureBase servers*) to authenticate connections from SecureBase clients.

- A SecureBase "drop-in replacement library" for the standard DB-Library. This library replaces a small number of DB-Library functions with SecureBase versions with the same names. Developers of DB-Library clients can link their applications against the drop-in replacement library to get SecureBase authentication without any source-code modifications.

- A mediator application which uses the other API libraries to do SecureBase authentication as either an inbound or outbound mediator (or both; for example, a mediator running on a firewall might do

Kerberos authentication of incoming connections and then establish an authenticated connection to a server on the other side of the firewall).

- Utilities for installing and administering the system.

SecureBase's complete detailed design is beyond the scope of this paper. However, the design of some of its components is illustrative of general techniques for layering strong authentication on top of third-party applications. These design components are discussed below.

7.1 The SecureBase protocol

The SecureBase protocol is implemented as a series of Open Server remote procedure calls made by the SecureBase client to authenticate and transmit necessary information to the SecureBase server. Like the Open Server protocol, the SecureBase protocol is query-response-based (i.e., all data transfer consists of a request from the client to the server followed by a response from the server to the client).

When a SecureBase client wants to log into a server, it follows these basic steps:

1. Connect to the server.

2. Transmit a Kerberos authenticator to the server.

   The Kerberos authenticator includes a block of data, given to the client by the Kerberos server, that can be decrypted only by the server. This data is what authenticates the client to the server. The authenticator also includes a session key which can be used for data encryption between the client and the server.

   The server responds with a message indicating whether the authenticator was accepted as valid and whether the client should transmit the client's passwords.

3. If requested by the server, transmit the client's password and/or remote passwords, encrypted using the session key contained in the authenticator.

   Since backward compatibility with existing clients and servers and a smooth migration path from old-style SYBASE security to SecureBase security are required, it may be necessary for Open Server authentication databases to remain in place for some period of time while SecureBase is being phased in. In that situation, SecureBase must interoperate with those authentication databases, which means, for example, that an inbound mediator will need to be given a client's password in addition to a Kerberos authenticator, so that the mediator can log into the Open Server for the client using the password. Similarly, remote passwords may be needed. Therefore, when a client transmits a Kerberos authenticator to the server, the data sent by the server in response includes flags telling the client whether or not to also transmit the client password and/or remote passwords. If requested, they are encrypted before transmission so they are not vulnerable to snooping.

   The server responds with an acknowledgement.

4. Tell the server that SecureBase authentication is complete. Again, the server responds with an acknowledgement.

These steps imply four message types that a client may transmit to a server: *authenticate, password, remote_passwords,* and *done.* Furthermore, one additional message, *change_password,* is supported. If a client wishes to change its password in the authentication database of a SQL Server protected by a mediator, but does not wish to do so using the SYBASE SQL `sp_password` stored procedure (which will cause his new and

Table 3: SecureBase protocol state table

		Client messages				
		authenticate	*password*	*remote_passwords*	*change_password*	*done*
States	**start**	`rd_auth` → **auth.**	→ **end**	→ **end**	→ **end**	→ **end**
	auth.	→ **end**	`rd_pw` → **auth.**	`rd_rpws` → **auth.**	`ch_pw` → **auth.**	`ck_done` → **done**

old passwords to be sent over the network in cleartext), it can send a *change_password* request to the mediator. This request contains the client's old and new passwords, encrypted in the session key for protection from snooping, and tells the mediator to change the client's password in the SQL Server's authentication database.

The protocol is summarized by the state table in Table 3. Valid messages are shown across the top of the table, and server states are shown down the left side. Each intersection shows the action that is performed (if any) and the state entered after the action. If the action fails, the server immediately jumps to the **end** state.

The **start** state is entered automatically when an incoming connection request is received. The **end** and **done** states are not shown because when the server enters either of those states, the protocol exchange has been completed (and therefore none of the messages listed in the state table can be received). The **end** state causes an immediate termination of the connection. The behavior of the server in the **done** state, which represents the end of authentication (rather than the end of the connection) is application-dependent. For example, an inbound mediator entering the **done** state will connect to the Open Server for which it is mediating and go into pass-through mode, whereas a SecureBase server entering that state will begin processing client data requests.

The behavior of the `rd_auth`, `rd_pw`, `rd_rpws` and `ch_pw` actions should be obvious. On the other hand, the `ck_done` action is perhaps less obvious. It verifies that the client has transmitted all necessary information (e.g., if the client password and remote passwords were requested, it verifies that they were sent), and if so, succeeds; otherwise, it fails, which causes the connection to terminate.

7.2 The SecureBase libraries

The three SecureBase libraries previously mentioned are `krbsybdb`, the client library; `krbsybsrv`, the server library; and `sybdb`, a drop-in replacement library for the standard Sybase DB-Library.

7.2.1 Client library

SecureBase clients link against the `krbsybdb` library in addition to Sybase's `sybdb` DB-Library. Low-level access to the internals of the SecureBase protocol are provided; however, most DB-Library clients will not have to utilize it, since their needs are met by the `krb_dbopen` function, which parallels, and takes the same arguments as, the DB-Library's `dbopen` function (in fact, `krb_dbopen` becomes `dbopen` in the drop-in replacement SecureBase library).

The `krb_dbopen` function extracts the authentication data of the client from the structure passed into it, and uses that data to authenticate to the server using the protocol described above.

7.2.2 Server library

SecureBase servers link against the `krbsybsrv` library in addition to Sybase's `srv` Server-Library. Because of the way Open Servers work, we cannot provide a server interface to SecureBase as transparent as the krb_dbopen client interface (nor can we provide a drop-in replacement for Sybase's Server-Library). In the current implementation, SecureBase servers must handle the steps of the authentication protocol themselves; i.e., they must support the state table in Table 3. Individual krbsybsrv functions are provided for reading an authenticator, reading the client's encrypted password, etc. However, a later version of SecureBase will probably provide more transparent server authentication.

7.2.3 Drop-in replacement library

The drop-in replacement library is constructed by merging the standard Sybase sybdb library, the Kerberos libraries (krb, des and com_err), and the krbsybdb library into a single library, a new version of sybdb. The names of several sybdb functions are changed in the library object files, and those functions are replaced with their krbsybdb counterparts. The new sybdb library can replace the old one and will appear identical to developers of DB-Library applications.

7.3 The SecureBase mediator

The basic functionality of the mediator application has already been discussed. A particular mediator invocation accepts incoming connection requests, either using Open Server authentication or SecureBase authentication, and then connects to another Open Server, again using Open Server authentication or SecureBase authentication. If all authentication is successful, the mediator enters pass-through mode for client and server and passes all further data transparently. This basic functionality is augmented by several features which may be interesting to designers of similar mediators for other applications.

7.3.1 Multiple outgoing authentication modes

A mediator invocation can choose whether or not to tell clients connecting to it to transmit their passwords and remote passwords. If they are requested, then they are used in the outgoing connection to connect to the desired Open Server (as described previously, the mediator should be running on the same machine as the Open Server, so that the passwords are not transmitted over the network in cleartext).

If passwords are not requested, the mediator can determine the password to use to connect to the Open Server in a number of different ways. In *SQL* mode, it connects to the desired SQL Server as the system administrator, asks it what the user's password is, and uses that password to log the user in. In *dynamic* mode, it does the same thing, but changes the user's password to a random, presumably unguessable password immediately after logging the user in. Dynamic mode is the recommended mode, since it makes the authentication database in the SQL Server obsolete, and at the same time makes the passwords in it unguessable, so all connections to the server must go through the mediator.

7.3.2 Trusted Kerberos principals

The mediator allows certain Kerberos principals to be treated as "superusers" by allowing any connection authenticated with one of the principals on the trusted access-control list (ACL) to specify any username when logging into the desired Open Server. This has a number of advantages; the two most significant are:

- Since all connections through the mediator can be logged, including the Kerberos principal making the connection and the username used to connect to the desired Open Server, a detailed audit log of administrative access to the Server can be produced. With standard Open Server authentication, the server knows only that someone logged in with the administrator username and password; if many people have the password, there is no accountability.

Table 4: Performance measurements of SecureBase

	Data transfer	Packet latency	Login latency
DB-Library client to SQL Server	1.00	1.00	1.00
DB-Library client to outbound mediator	2.58	1.10	1.20
SecureBase client to inbound mediator	2.23	1.32	1.33

- Since administrative access is controlled by an ACL, the administrator password does not have to be given out, and administrator access for a particular individual can be revoked by editing the ACL rather than by changing the administrator password and distributing it to everyone else.

8 Performance issues

We measured the effect of SecureBase authentication on Open Server connections by measuring the time required to perform the following operations:

- Transferring a large amount of data from a SQL Server to a client (data transfer rate).
- Transmitting a short query to a SQL server and receiving a short response (packet latency).
- Logging into a SQL server (login latency).

We performed each of these operations in three different connection modes:

- A direct connection from a standard DB-Library client to a standard SQL Server.
- A connection from a standard DB-Library client to an outbound mediator to an inbound mediator to a standard SQL server.
- A connection from a SecureBase client to an inbound mediator to a standard SQL server.

Our results are shown in Table 4, normalized to the control case (standard DB-Library connection to a standard SQL server). In addition, for the SecureBase client, we performed an additional login test in which an existing Kerberos ticket cache was used for all logins, rather than obtaining new tickets from the Kerberos server for each login. The performance improvement was negligible.

Our performance testing confirmed our hypotheses that transmitting data through two additional processes (i.e., the outbound and inbound mediators) would cause significant performance degradation, and that reducing the number of additional processes to one (by connecting directly to the inbound mediator) reduces, but does not eliminate, the performance degradation.

However, some of the results of our testing were surprising to us, and we cannot adequately explain them. Why should the latency of a single packet of data be greater when going through one mediator than when going through two? Similarly, why does it take longer to log in when going through one mediator than when going through two? Finally, why was there no noticeable performance improvement when we used an existing Kerberos ticket cache for all logins instead of getting new tickets each time?

It is possible that we need to modify our testing environment in order to better emulate production use of SecureBase. For example, our Kerberos server is on the same machine as our SQL server, and all testing took place on a single lightly loaded ethernet segment. Loading the test network more, placing the servers on different machines, and placing the servers and clients on different subnets might tell us more. Furthermore, we need to do a more detailed analysis of where the most time is actually being spent.

9 Generalizing the technique

Although SecureBase's code is not easily portable to other applications besides SYBASE, because it uses the Open Server libraries extensively, the general concepts applied by SecureBase toward the problem of securing SYBASE are easily generalizable to other relational database systems and other client-server applications.

Any attempt to add strong authentication to an existing network application should follow the steps listed in Section 2. The following sections discuss a general approach to some of those steps.

9.1 Locating security problems

The set of potential security problems for a network application is relatively small. Since SYBASE is relatively wide open to network attacks on its security, the problems identified in Section 3 (i.e., password theft, data theft, data modification, unauthorized password use, and independent authentication databases) can be used as a "base set" when examining other applications. It is usually quite straightforward to determine if an application is vulnerable to one of these attacks — unless the documentation for the application says otherwise, it *probably* is.

9.2 Listing potential solution techniques

The techniques proposed for securing SYBASE (in Section 4) are a good base set to consider when analyzing how to secure another application. However, some of those techniques may not be applicable in all cases. Here's what to look for when considering them:

Securing the network. The applicability of securing the network is independent of the application, since physical network security is (mostly) independent of the applications run on it. Instead, consider the feasibility of securing the network, given the problems outlined in Section 4.1.

Application authentication. Do you have access to the source code for the both the client(s) and server(s) to which you want to add authentication? If they use a vendor-provided API, can you use that API to transmit and receive arbitrary authentication data (encoded, if necessary, in a form that the API can transmit)?

Application encryption. In addition to all of the requirements for application authentication, do you have access to the data stream at a low enough level that encrypting it will not interfere with the application protocol? If not, do you have access to the specifications for the application protocol, and can you handle necessary data conversions yourself so that the vendor API doesn't have to?

Protocol-level mediator authentication. Are there vendor-provided APIs for both clients and servers? Can they interoperate in the same process, i.e., can a process be both a client and a server? Is there support for a server gatewaying between a client and another server? Is there a way to trick clients and servers so that connections are routed through your mediator? As for application authentication, can you transmit arbitrary authentication data within the bounds of the protocol?

Protocol-level mediator encryption. In addition to all of the requirements for protocol-level mediator authentication, can you encrypt the data stream safely as for application encryption?

Socket-level mediator authentication. Can you extract authentication data from a TCP stream? Is the format of authentication data in the stream likely to change from version to version of the application protocol? Are you going to have to support different network types? Can you trick clients and servers into rerouting connections?

Socket-level mediator encryption. Are you going to have to support different network types? Can you trick clients and servers into rerouting connections?

When listing potential solution techniques, it is important to spell out the problems each technique solves and its advantages and disadvantages. This greatly simplifies the tasks of ranking and choosing the techniques to use.

9.3 Ranking problems, advantages and disadvantages

The ranking of problems, advantages and disadvantages should answer several questions. Which problems are important to solve, and which can be ignored or left for later? How important is having each advantage? How important is *not* having each disadvantage? Implicitly or explicitly, the rankings state the requirements of the desired solution.

Someone setting out to improve the security of an application generally has in mind some problems which *must* be solved. Therefore, a good first ranking step is to establish a category for definite requirements and decide what to put in it. Then, decide how much granularity is needed when ranking the non-requirements.

Often, a simple "requirements" vs. "desirables" ranking, such as the one used for SecureBase, will be sufficient. However, if more precision is needed (e.g., if it's necessary to approach a manager and ask for a decision about which approach to take, and he wants the values of the various approaches quantified), it may be necessary to rank the desirables more precisely, e.g., on a numerical scale.

9.4 Choosing appropriate techniques

The ranking should make it easy to choose which technique(s) to use. If it doesn't, then you should redo the ranking until it does. The "ranking" step should in fact be considered part of the "choosing" step, and should be the larger part of it.

If one technique meets all of the requirements specified by the rankings, then use it. Otherwise, figure out how multiple techniques can be integrated to meet the requirements together.

10 Future work

We hope to develop the ideas presented in this paper further in a number of different areas:

- Adding support for data encryption and/or integrity verification.
- Porting to other RDBMS's.
- Porting the API libraries to non-UNIX platforms.
- Exploring some of the other techniques described above, e.g., using socket-level mediation instead of protocol-level mediation.
- Developing a more generalized approach which avoids the problems described above, so that it is easier to add authentication to other applications.
- Porting to other network applications.

Furthermore, we hope to improve our performance analysis of SecureBase, as described in Section 8.

References

[1] Open Software Foundation. *Application Development Reference*, volume 1 of *OSF DCE 1.0 documentation set*. Open Software Foundation, update 1.0.1 edition, July 1992.

[2] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authenticaion Service for Open Network Systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February 1988.

[3] Sun Microsystems, Inc. XDR: External Data Representation Standard. RFC 1014, Network Working Group of the Internet Engineering Task Force, June 1987.

[4] J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 232–244, Oakland, California, May 1991.

[5] G. Winfield Treese and Alec Wolman. X Through the Firewall, and Other Application Relays. In *Usenix Conference Proceedings*, pages 87–99, Cincinnati, Ohio, June 1993.

Dial-In Security Firewall Software

Bob Baldwin
Los Altos Technologies, Inc.
2111 Grant Road, Los Altos CA 94024
baldwin@lat.com

Jim Kubon
Sun Microsystems Computer Corporation
2550 Garcia Ave., MTV 23-204 Mountain View CA 94043
jimk@eng.sun.com

Abstract

This paper describes how Sun Microsystems Computer Corporation uses a dial-in security firewall product to protect it's systems and networks from modem wielding computer crackers. Specific examples show how LAT TermServ was configured to provide strong authentication, controlled access to the network, high performance, and management reports that helped with troubleshooting and capacity planning. Other examples show the difficulties of managing modem and network access in a large and rapidly growing company like Sun.

Introduction

Internally, Sun systems are well connected by high speed networks. Ideally, all of Sun's sites and the homes of all Sun employee's would be directly attached with high speed networks, but that is not economical. In many cases, the frequency and duration of connections cannot justify the cost of a high speed dedicated link.

The obvious solution is to use modems and the public switched network to provide on-demand access to the Sun internal network. This solution takes advantage of the low price of both modems and telephone connect time. The remainder of this paper explains how a "dial-in security firewall" program helped solve the security and management problems that arise with this solution in much the same way that an Internet firewall solves the problems that arise when wide area networks are used to provide access.

The first part of this paper describes the basic goals and features of the LAT TermServ program, while the second part discusses the challenges of running this kind of software in a large and rapidly growing company. The first author works for the company that owns the LAT TermServ program, and the second author is responsible for a large TermServ installation at Sun.

Features of a Dial-In Firewall

A dial-in firewall must be able to operate in two different modes. In the simplest case, it protects a single stand-alone machine. More generally, it acts as a modem-pool server that provides controlled access to a network of machines.

Goals for a Dial-In Firewall:

1. Strong authentication and accountability.
2. A solution that could be used widely within the company.
3. High performance, low costs.
4. Controlled access to network resources.
5. Support for all services like UUCP, X-terminals, and PPP.
6. Modem pool management and usage reports.

To protect company resources, the system uses extra passwords and dial-back to authenticate the user. The extra passwords help thwart call-forwarding attacks and attacks on sites that use the same line for dial-in and callback. Sun encourages sites to configure the program to require one password when the user dials-in, and another password when the system calls back.

Callback is time consuming, but it has other benefits beyond security. It provides an easy way for Sun to pay for employee phone charges, and the management reports can be used to bill individual departments if need be.

TermServ can offer the user a choice of numbers to callback if they have multiple places where they work, and each place can have different configuration options to describe a dumb terminal, an X-terminal, or a workstation. When an employee is at a conference, their account can be configured to let them type in the number to callback, or to skip the call-back altogether. Clearly the security risk of such a set up must be weighted against the benefits of access.

Once a user has gone through the time-consuming process of call-back authentication, the program can use the rlogin protocol to avoid the need for further passwords, at least for the computers that trust the dial-in firewall.

When a cracker does compromise an account, the investigation is aided by the link to a phone number and by the program's ability to record both ends of the terminal session whenever that specific user logs in.

A company-wide solution was essential to stem the growing security hole that was being widened when each new site added modems to its computers. Individual users and sites did not want to purchase dialback modem hardware, whereas they were willing to run software that also helped with modem management and troubleshooting. To simplify accounting and distribution, Sun purchased a global license. Individual sites just copy the program from the main file servers.

The goal of high performance helped reduce the overall costs by allowing Sun to use older CPUs as the dial-in firewall machines. The rlogin and telnet programs included with TermServ use vastly less CPU time than the standard programs. Thus, an old CPU could act as a network terminal concentrator for eight or more modems. Table 1 illustrates the performance difference for a 9600 baud connection running on a Sun 4/110. The performance improvement is primarily due to implementing the rlogin as a single processes with the "select" asynchronous I/O interface, as opposed to the pair of synchronous processes used by the standard rlogin program.

Table 1: LAT TermServ rlogin Performance

	% CPU used (9600 baud, Sun 4/110)
Standard rlogin	4.1
TermServ rlogin	0.2

Even with strong authentication, the system must place restrictions on the scope of the allowed network access. The program can restrict users to a specific list of hosts or list of subnets. Restricting network access can achieve administrative as well as security goals. For example, the users of the management modem pool can be restricted to management systems in order to prevent engineers from using the management modem pool.

Once users are authenticated, they need access to the full range of network services. The program comes with interfaces for rlogin and telnet, and a hook for UUCP that is explained below. TermServ can be configured to offer access to other front-end programs that the site specifies. For example, one of Sun's PPP servers is implemented in this way. Alternatively, connections can be made in 8-bit binary mode to a remote host that runs a driver for X windows, SLIP or PPP.

The system was designed to work with UUCP. In fact, when a UUCP call comes in, it is possible to have the connection refused, and then the dial-in firewall machine automatically start a UUCP call to pickup mail and netnews from that remote site. That site does not need to know that TermServ is being used as long as it will accept the call. Of course, if the remote UUCP site is also running TermServ, it should be set up to accept the connection without further call-back, otherwise the two machines will spend all day refusing each other's callback attempts.

Setting up modems for dial-out access on Unix is time consuming for the inexperienced, and the program does not help with that process. However, once the modems are set up, the program can help with trouble shooting and capacity planning. It keeps a detailed log of modem activity that is automatically reduced each night by a cron job that mails a summary report to the administrator. There is also a weekly highlight report. A broken modem shows up quickly because the per-modem report shows that there has been no activity on that modem. Additional reports can be invoked from the command line to display other information such as a histogram of modem usage by time of day, which can be used to evaluate the need to purchase extra modems or faster modems.

By saving the daily reports in a single file, it is possible to quickly determine when a user last logged in by grep'ing for the username. Some sites have written a shell script to identify all the users who have not logged in for a long time, and thus are good candidates for having their access removed.

Discussion

A first look at LAT TermServ seems to do everything that could be wanted from a dialback software package; using it confirms this first opinion. It's fast, reliable, and very full featured. It has a plethora of options that perform a wide variety of tasks, from dialback to performing keystroke logging (or "tapping") of calls and more.

Unfortunately, under closer examination, we found that TermServ might be <u>too</u> full featured. Like other software packages for Unix, the power and flexibility of the software can lead to unexpected security problems. Most of Sun's systems with modems only need the dialback feature to conform to Sun's internal security policies. Many of the program's features can undermine this policy. The configuration files can only be modified by `root`, but if system crackers have become `root`, they can modify the configuration to leave back-doors. In particular back-doors can be installed by:

1. Disabling dial-back for a particular account.
2. Creating a new account.
3. Removing a modem from LAT TermServ control.
4. Invoking an arbitrary program running as root.

These are the same back-doors that can be installed by changing the `/etc/passwd` file. All these backdoors are simple to recognize, so Sun has augmented its security assessment shell scripts to verify that the dial-in firewall machines are configured correctly.

Although the manual is fairly good, Sun's experience is that most administrators of a package rarely look at more than the installation section (if that), and will add features as necessary either by emulating another friendly site's configuration files or by reading just enough of the manual to get the desired feature to work, without regard to the consequences. Oddly enough, the online man pages were good enough for one system cracker to figure out how to leave some of the back-doors mentioned above,. However, some system crackers don't read the whole manual either. At another company using LAT TermServ, a system cracker broke in via the network, achieved root access, ftp'ed the man pages for the program and learned enough to create a new account that would allow him to be called back at any number he specified. What he didn't understand was that the program keeps a log file that records the callback number, so it was easy to identify the cracker.

In Sun's rapidly growing network environment, the program's network restrictions are not very convenient. New subnets are being added each week and in theory the LAT TermServ database would have to be updated each week. The program does allow wildcards in the subnet specification, which solved the problem until the engineering group exceeded 255 subnets. Over time, some subnet numbers have also been reassigned to other parts of Sun, so a simple wildcard does not express the desired policy. However, host and subnet restriction work well for contractors and other non-employees that have very specific access needs.

Conclusion

The technology for a software-only dial-in security firewall exists today and is quite usable in its current form. A dial-in security firewall is a useful complement to an internet firewall system for environments that need to provide occasional network access to remote sites in a secure manner.

The main features that would enhance the security and usability of the system would be an interface to smart-card tokens and a kerberos version of rlogin. A graphical interface would make it easier for system administrators to install and manage, as would sample shell scripts that users could run to request dial-in access and specify phone numbers.

Secure RPC Authentication (SRA) for TELNET and FTP

David R. Safford, David K. Hess, and Douglas Lee Schales
Supercomputer Center
Texas A&M University
College Station, TX 77843-3363

1. Abstract

TELNET and FTP currently exchange user authentication (passwords) in plain text, which is easily eavesdropped. Several techniques, such as Kerberos and SPX, have been proposed in draft RFCs to implement secure authentication. These techniques, however, have several drawbacks, including technical complexity, poor vendor support, and organizational problems. This paper presents SRA, a very simple and tested technique based on Secure RPC which, while certainly not as strong as RSA, is reasonably strong, fast, and trivial to implement immediately for both inter and intra-domain communication.

2. Background

TELNET and FTP currently pass the user authentication across the network in the form of plaintext passwords. These passwords can trivially be eavesdropped with such simple tools as etherfind and tcpdump. During intrusions at Texas A&M University in August 1992, significant amounts of the tools used by the crackers were captured. They had much better tools than simple ones such as these, and they used them to capture passwords. It is absolutely essential that TELNET and FTP be extended to provide confidential user authentication to prevent such simple password grabbing.

Several RFCs and draft RFCs address this issue, including:

1409 (TELNET Authentication)
1411 (Kerberos Authentication)
draft-ietf-telnet-authspx-00.txt
draft-ietf-cat-ftpsec-01.txt

These drafts outline proposed designs for the use of Kerberos, RSA, SPX, and gssapi for confidential authentication. While these proposed techniques afford excellent security, they suffer from some drawbacks which have kept them from widespread use so far. Kerberos is difficult to implement and requires centralized ticket servers with copies of all user passwords in plaintext, which is possibly undesirable in large organizations with relatively autonomous subgroups. Inter-domain kerberos is particularly complex. Also, vendors have been extremely slow to provide commercial support for kerberos. Various public key methods can eliminate the need for centralized secret key storage and can solve inter-domain issues very nicely, but need a secure public key distribution system, which will take time to popularize. This paper proposes a simple confidential authentication system for TELNET and FTP based on Secure RPC techniques.

The proposed method has several advantages:

1. The user authentication information is sent encrypted with a one-time DES key.
2. The method uses one-time random Secure RPC keys and, therefore, needs no external keyservers or complex key protection.
3. The method is compatible with all existing and draft authentication methods, so it can take advantage of the method, if supported, transparently on the user's behalf.

4. The method is reasonably fast (it adds about 1 second to a connection on a sparcstation 1 class machine).
5. The method is easily implemented entirely within the client and server binaries, so drop-in installation is very simple. No external key distribution service is needed.
6. All source code is publicly available, making support available for most platforms, independent of vendor support.

One disadvantage of the method is that, unlike more comprehensive public key methods, it does not provide mutual repudiation. The server does validate the user's identity, but the client does not perform server authentication. A second weakness is that Secure RPC is subject to logarithm attacks [1], although such attacks are not trivial. The proposed design for Telnet and FTP reduces this exposure by computing a new random secret/public key pair for each connection. A third possible limitation is that the underlying Diffie-Hellman algorithm is patented. As most vendors already offer Secure RPC as part of Secure NFS implementations, they have already made the necessary key service available.

The bottom line is that the method is significantly more secure than sending passwords in plain text, while offering greater ease of implementation than other, more secure, alternatives.

3. Design

Secure RPC uses a simple version of the Diffie-Hellman exponential based method for determining a common DES key. This algorithm and its specific implementation parameters for Secure RPC are given in the RFC for RPC (RFC 1057). Normally Secure RPC stores a user's key in the keyserver, encrypted with the user's password, and uses it as necessary to calculate common keys for each connection. In the proposed method, this is greatly simplified by calculating a new random key set for each TELNET or FTP authentication, which eliminates the need for any keyserver. Thus the Secure RPC code is used only for performing basic key calculations, not for key storage or management. The basic data flow is shown in figure 1. The functions performed by the Secure RPC code are indicated by brackets, and the parentheses indicate DES encryption or decryption.

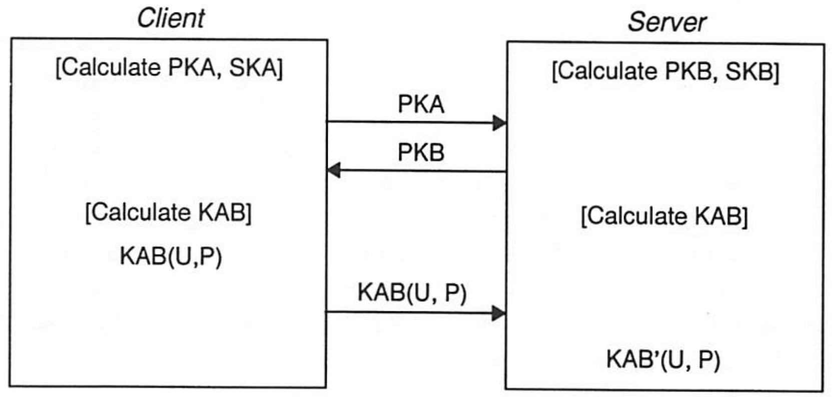


Figure 1. SRA data flow.

From the public Secure RPC code, the keys are 192 bit integers calculated as:

SK (secret key) = random 192 bit number

PK (public key) = (base^{SK}) mod modulus

where base = 3

modulus = `d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b`

Since exponentiation is commutative, given the key pairs (PKA, SKA) and (PKB, SKB), both sides can confidentially calculate KAB as:

Server: KAB = (PKA^{SKB}) mod modulus

or

Client: KAB = (PKB^{SKA}) mod modulus

For purposes of the DES encryption, Secure RPC uses the middle 64 bits of the 192 bit common key (KAB). The calculation of KAB is reasonably secure, as it requires at least one of the secret keys, which are not put on the network, or calculating the logarithm of large numbers.

4. Incorporation into TELNET

The design for incorporation into TELNET followed exactly the design specified by the respective RFC for Kerberos, which is already implemented in the Network Release 2 code. The RFC specifies the necessary option negotiation (WILL, WONT, DO, DONT) to ensure that both sides are able to handle the authentication method. If not, the exchange reverts to the default unencrypted transfer.

SRA has five suboption commands: KEY, USER, PASS, ACCEPT, and REJECT, which are demonstrated in the figure 2.

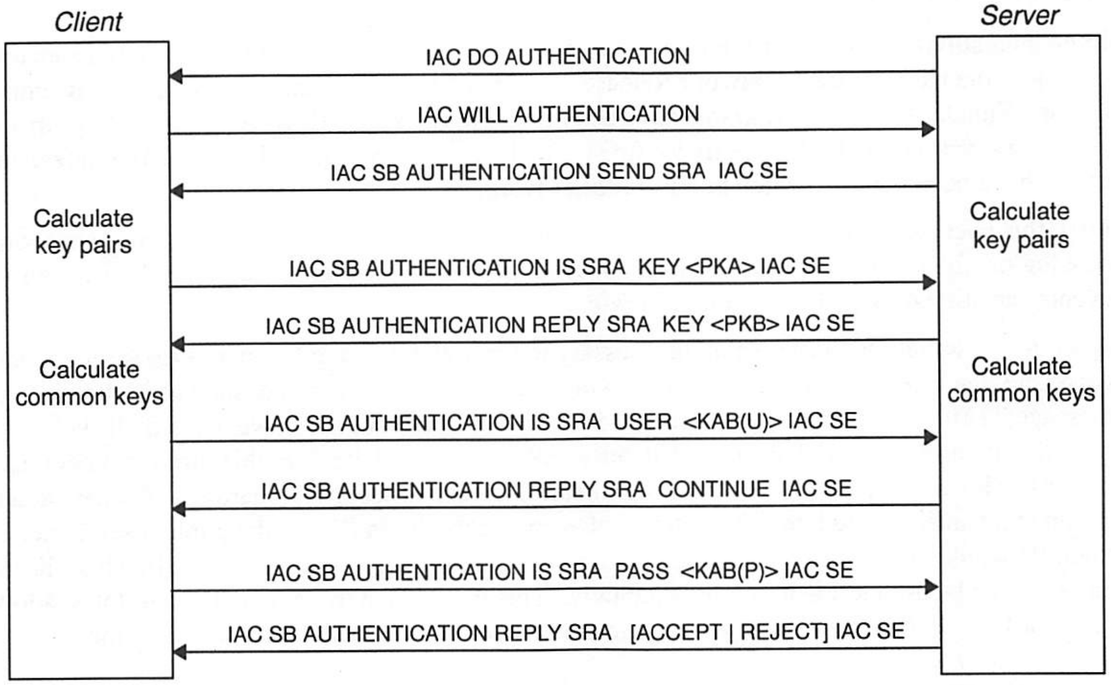

Client
Server
IAC DO AUTHENTICATION
IAC WILL AUTHENTICATION
IAC SB AUTHENTICATION SEND SRA IAC SE
Calculate key pairs
Calculate key pairs
IAC SB AUTHENTICATION IS SRA KEY <PKA> IAC SE
IAC SB AUTHENTICATION REPLY SRA KEY <PKB> IAC SE
Calculate common keys
Calculate common keys
IAC SB AUTHENTICATION IS SRA USER <KAB(U)> IAC SE
IAC SB AUTHENTICATION REPLY SRA CONTINUE IAC SE
IAC SB AUTHENTICATION IS SRA PASS <KAB(P)> IAC SE
IAC SB AUTHENTICATION REPLY SRA [ACCEPT | REJECT] IAC SE

Figure 2. Sample nominal authentication data flow.

5. Incorporation into FTP

The FTP incorporation similarly follows the draft RFC specification for the kerberos exchange (see figure 3), although in this case, the kerberos code was not already implemented in the Network

Release-2.

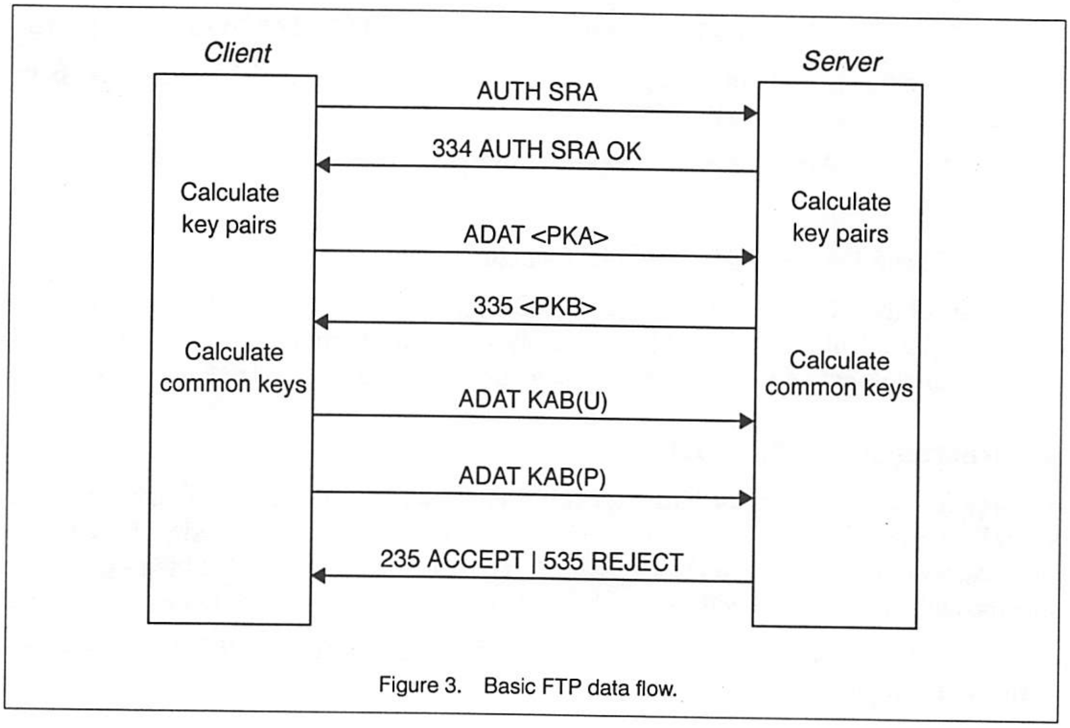


Figure 3. Basic FTP data flow.

6. Preliminary Results

Implementations for both TELNET (telnet and telnetd) and FTP (ftp and ftpd) have been incorporated into the baseline BSD Network Release-2 code and tested on various sparcstation machines running SunOS 4.1.2. The programs have been found to interact well with every existing ftp and telnet service tried, including the major ftp sites, which use experimental servers. Two interesting things have been noticed, which point to further work.

First, this encryption prevents network monitoring programs, such as *Etherscan*, from observing root logins, ftp's and other knob-turning. Overall, this is not too significant a problem, as these events can still be logged by the target service.

A second observation concerns the unnecessary tendency of existing telnet and ftp server code to echo back the user name in plain text following successful authentication, such as in the ftp return message "`230 User smith logged in`". In our implementation we deleted all such direct user identification, although the user's identity will probably still be available through other means, such as rwho and finger. It would be a significant increase in security, regardless of what authentication mechanism is used, to differentiate between a secret login ID and the public user name. The login ID would only be used for authentication purposes (along with the password), while the user name would be used to identify a user publicly. This would be a trivial extension to the shadow password concept, and would be easily implemented on top of the SRA implementation.

7. Availability

The complete source packages for both TELNET and FTP, including all necessary client and server code, (but not including the Secure RPC code), is available on:

sc.tamu.edu:pub/security/SRA.

The relevant Secure RPC code is available in:

ftp.uu.net:networking/rpc/tirpcsrc.tar.Z,

keyserv/setkey.c (routines to calculate common key)

keyserv/generic.c (routines to generate public/secret key pair)

8. References

[1] Brian A. LaMacchia and Andrew M. Odlyzko. "Computation of Discrete Logarithms in Prime Fields", Designs, Codes, and Cryptography, volume 1, 1991, pp46-62.

[2] RFC 1057 "RPC: Remote Procedure Call Protocol Specification, Version 2.", ftp.uu.net:/inet/rfc/rfc1057.Z

Caller Identification System in the Internet Environment

Hyun Tae Jung, Hae Lyong Kim, Yang Min Seo
Ghun Choe, Sang Lyul Min, Chong Sang Kim, Kern Koh

Dept. of Computer Engineering, Education and Research Computing Center
Seoul National University
San 56-1, Shinlim-dong
Kwanak-gu, Seoul, 151-742
Korea

Abstract

The Caller Identification System is a security system designed to strengthen the security measures already present in the SNU(Seoul National University) local area network. It is a system made up of two main components and they are (1) an Extended TCP Wrapper(*ETCPW*) and (2) a Caller Identification Server(*CIS*). The Extended TCP Wrapper(*ETCPW*) is, as its name suggests, a TCP Wrapper[1] with extra features added to it. The original TCP Wrapper was able to store information only about the immediate requester machine. The *ETCPW*, on the other hand, records and provides information on the complete route (starting from home-base to last machine used) taken by each user in the system. The other component, that is *CIS*, provides the network route of each user to the *ETCPW* so that *ETCPW* can check that user's identity. The one great advantage of the Caller Identification System is that if a cracker has managed to invade a system, despite the added security measures, we are able to retrieve information (using post-mortem analysis) on that hacker's route, which will eventually lead us to his or her home machine.

1 Introduction

Like most security systems, our "Caller Identification System" started out with a "hacker" who frequently entered our LAN, created mischief, hid his/her trail and then skipped out leaving us with a big problem that would take us hours to solve. Therefore, because of our "friend", we were assigned to draw up a stricter security system for the LAN within the SNU (Seoul National University) campus. After several weeks of discussion, we came up with a method that involved (1) extending the features of the *TCP* Wrapper within the system and (2) creating a server called Caller Identification Server (*CIS*) that is used to identify the home machine of a network requester.

The *TCP* Wrapper[1] is basically an authentication program. This program is used as an authentication control that restricts access from certain hosts or network. The data provided by the original *TCP* Wrapper in our University's LAN only had information about the immediate requester machine. This made it difficult for us to trace the routes the hacker had taken. To rectify this problem, we have extended the functions of the *TCP*

Wrapper in our system. This newly enhanced *TCP* Wrapper, which we call Extended TCP Wrapper, (*ETCPW*) not only provides information on the immediate requester machine but also on requester machines prior to it (i.e. the chain of machines from the machine of commencement to the present machine).

The Caller Identification Server installed on each machine provides information, concerning the network route taken so far, to the *ETCPW*. *ETCPW* then utilizes this information for logging and authentication purposes. Should a cracker's attempt at hacking be successful, the combination of the information provided by the above *ETCPW* and *CIS* will make it possible for the system administrator to identify at least one cracked machine. This, in turn, enables us to trace a culprit to his or her home-base using post-mortem analysis.

Our system shares many similar functions to that of the RFC1413 Identification Protocol[2] (RFC1413 is used to identify a caller on a particular TCP connection). The one distinct feature that separates our system from RFC1413 is that the RFC1413 only allows each machine to identify the immediate requester machine while our system has information on the entire path of a user on each machine in that path.

2 Caller Identification System

In most UNIX-based *TCP* implementations, a request for a specific program is first processed through inetd. The system then runs the predefined server program (Figure 1a). In a system enhanced by the *TCP* Wrapper, (Figure 1b), any access to a network service has to go through the *TCP* Wrapper program which is situated between the inetd and server programs. The *TCP* Wrapper does filter-out, logging and takes other security measures. Any access to a network using this system involves filter-out, logging and other security measures. This, however, has proved insufficient, so our Caller Identification System was created. Our Caller Identification System, as shown in Figure 1c, contains all the functions provided by the *TCP* Wrapper plus extra functions where a caller has to go through authentication in order to access the network service. This "new" *TCP* Wrapper called *ETCPW* (Extended TCP Wrapper), makes use of information provided by the Caller Identification Server (*CIS*). The *CIS* on each machine stores the complete network trace of all users of that machine. In short, the *CIS* receives requests from the *ETCPW*, sends requests to the *CIS*s of other machines for authentication, receives responses from them and passes these responses to the *ETCPW*.

2.1 Basic Idea

A "hacker" may deliberately go through a complicated path (using login several times) to get to his/her desired destination. He/She does this to hide his/her home-base. The hacker who has successfully invaded his/her desired target machine will then be able to "corrupt" it. When this happens, it will be up to the system administrator to try to trace the path of the hacker back to his/her home-base. This, however, would prove to be extremely difficult to do if the "hacker" had taken a long and complicated path. To make matters worse, this tracing requires information on each intermediate machine which may not be

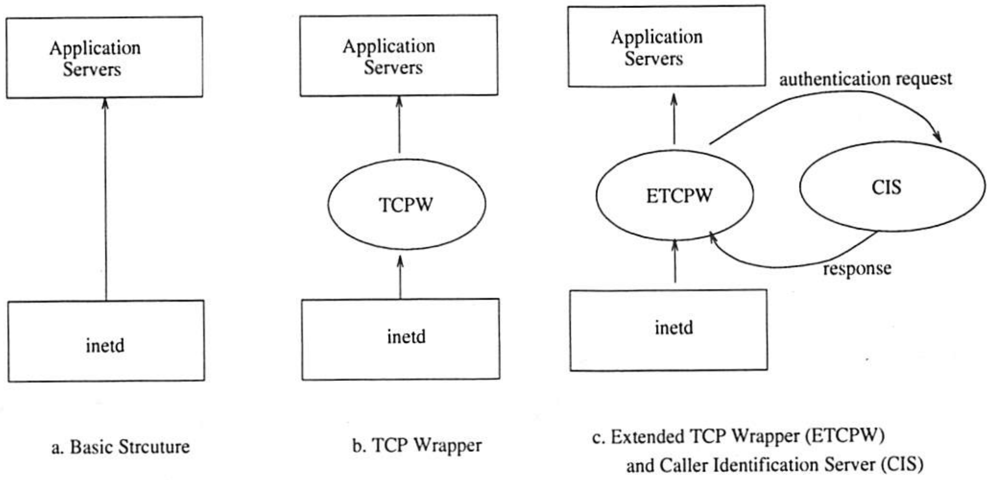


Figure 1: ETCPW and CIS

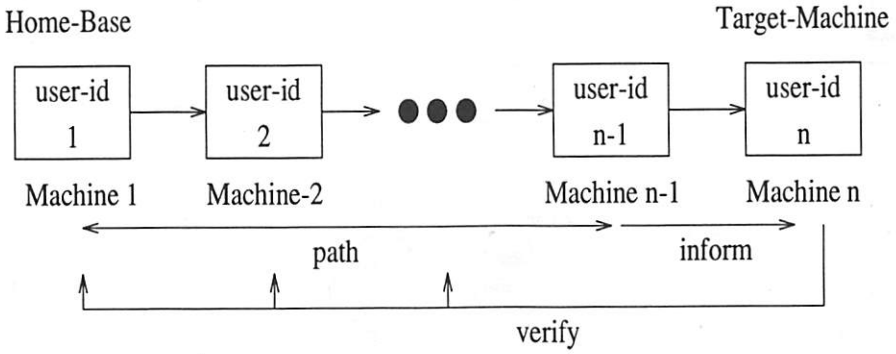


Figure 2: Network Trace

readily available to the administrator. Our system was created specifically to overcome this particular handicap.

As an example of how our system works let us consider Figure 2. Figure 2 shows a situation, where a user of home-base $Machine_1$ tries to get permission to access $Machine_n$ with $user\text{-}id_n$. Here, the user has already traversed the path from $Machine_1$ to $Machine_{n-1}$. When the user asks for permission to access $Machine_n$, $Machine_n$ will send a request to $Machine_{n-1}$ for the user's path (home-base to the $Machine_{n-1}$). $Machine_{n-1}$ then sends the path the user has taken, back to $Machine_n$. $Machine_n$ then tries to verify that path by sending Authentication Requests to all machines in that path. Only after receiving positive Authentication Responses from all those machines can authorization be granted to this user to access $Machine_n$. This complete information in each machine of each user helps to trace a hacker's path more easily.

2.2 Caller Identification Procedure

Figure 3 shows the process carried out to identify a caller/user when the user with the path $Machine_1....Machine_{n-1}$ wishes to access $Machine_n$ (its destination). The following

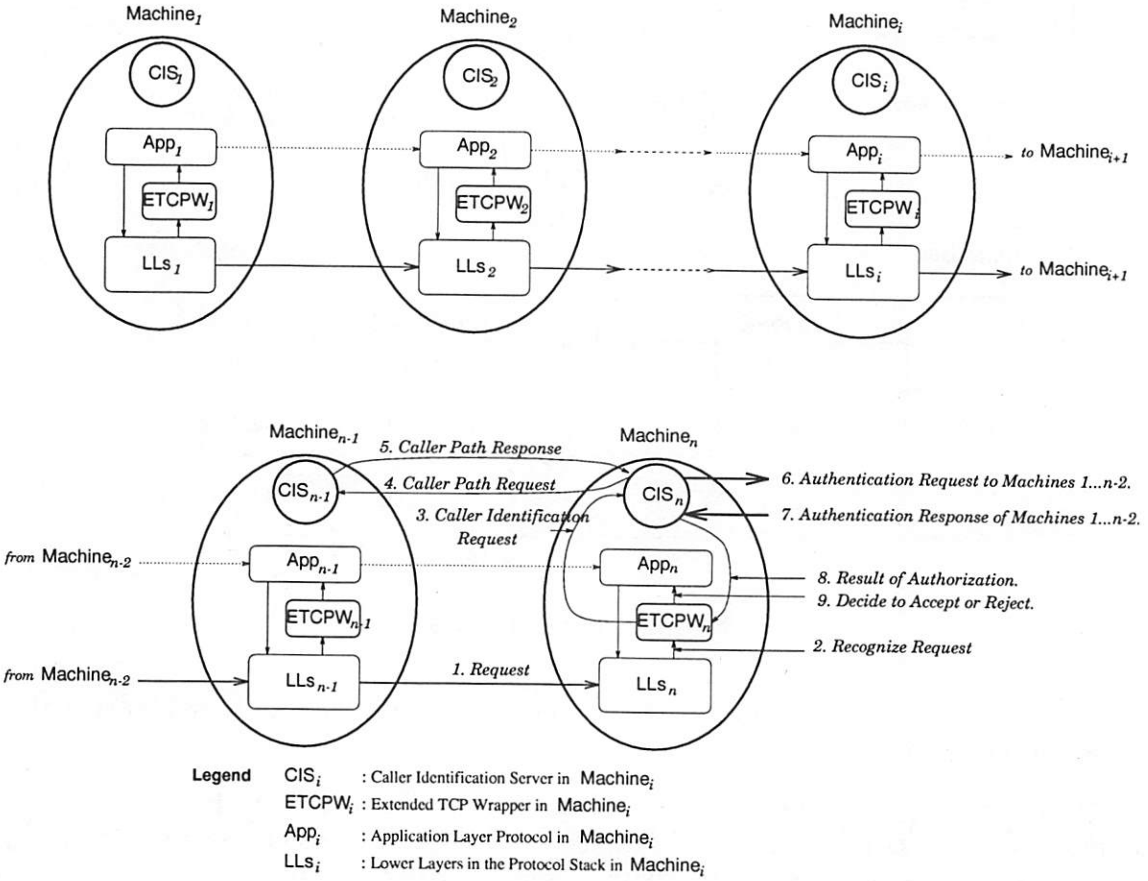

Machine$_1$
Machine$_2$
Machine$_i$
CIS$_1$
CIS$_2$
CIS$_i$
App$_1$
App$_2$
App$_i$
to Machine$_{i+1}$
ETCPW$_1$
ETCPW$_2$
ETCPW$_i$
LLs$_1$
LLs$_2$
LLs$_i$
to Machine$_{i+1}$
Machine$_{n-1}$
Machine$_n$
5. Caller Path Response
4. Caller Path Request
3. Caller Identification Request
CIS$_{n-1}$
CIS$_n$
6. Authentication Request to Machines 1...n-2.
7. Authentication Response of Machines 1...n-2.
from Machine$_{n-2}$
App$_{n-1}$
App$_n$
8. Result of Authorization.
9. Decide to Accept or Reject.
ETCPW$_{n-1}$
ETCPW$_n$
2. Recognize Request
from Machine$_{n-2}$
LLs$_{n-1}$
1. Request
LLs$_n$
Legend
CIS$_i$: Caller Identification Server in Machine$_i$
ETCPW$_i$: Extended TCP Wrapper in Machine$_i$
App$_i$: Application Layer Protocol in Machine$_i$
LLs$_i$: Lower Layers in the Protocol Stack in Machine$_i$

Figure 3: Sequence of Events of the Caller Identification System

details the steps the Caller Identification System has to go through in order to successfully establish the connection desired (i.e., access *Machine*$_n$).

1. *Request*: The Application Layer Protocol of *Machine*$_{n-1}$ sends a request through the Lower Layers of *Machine*$_{n-1}$ to the Lower Layers of *Machine*$_n$.
2. *Recognize Request*: The request received from *Machine*$_{n-1}$ by the Lower Layers of *Machine*$_n$ is sent to the `inetd` and then the *ETCPW*. *ETCPW* then analyzes the request.
3. *Caller Identification Request to CIS*$_n$: After analyzing the request *ETCPW* then sends a Caller Identification Request to *CIS*$_n$.
4. *Caller Path Request to CIS*$_{n-1}$: *CIS*$_n$ then establishes a connection with the *CIS*$_{n-1}$ of *Machine*$_{n-1}$ and sends a Caller Path Request to it.
5. *Caller Path Response from CIS*$_{n-1}$: After having received the Caller Path Request from *Machine*$_n$, *CIS*$_{n-1}$ then looks within the internal table to decide whether the login process that made the network request is from a remote or local machine. If the request is associated with a remote login, the network trace taken by the remote login, which is kept in a table maintained by *CIS*$_{n-1}$, is sent as a Caller Path Response to *CIS*$_n$. Otherwise (i.e., if the login is from a local terminal) it will send to *CIS*$_n$ a Caller Path Response that will indicate that *Machine*$_{n-1}$ is the home base machine of the requester.
6. *Authentication Requests to Machine*$_1$*...Machine*$_{n-2}$: Using the network trace or route it has received from *CIS*$_{n-1}$, *CIS*$_n$ can then respond by sending Authentication Requests to all the machines mentioned in the network trace. This Authentication Request is done to verify that the network trace sent by *CIS*$_{n-1}$ is genuine.
7. *Authentication Responses from CIS*$_1$*...CIS*$_{n-2}$: *CIS*$_1$*...CIS*$_{n-2}$ receive the Authentication Request from *CIS*$_n$ and check their tables. If the path exists in their tables then they respond positively, else they respond with a "no".
8. *Result of Authentication*: If it receives positive responses from *CIS*$_1$*...CIS*$_{n-2}$, *CIS*$_n$ will then be inserting into its own table information on the recent connection made (i.e., connection to *Machine*$_n$) and logging it. It then tells *ETCPW* that authentication has proceeded successfully and sends a Caller Identification Response of "yes" to the *ETCPW*. If even one of the machines (*Machine*$_1$ to *Machine*$_{n-2}$) gives a negative response, *CIS*$_n$ will send a "no" response to *ETCPW*.
9. *Permission to Connect*: Depending on the response from *CIS*$_n$, the *ETCPW* decides whether to accept or reject a connection request.

2.3 *CIS* and *ETCPW*

The Caller Identification Server, as shown in Figure 4, sends and receives three types of request/response packets.

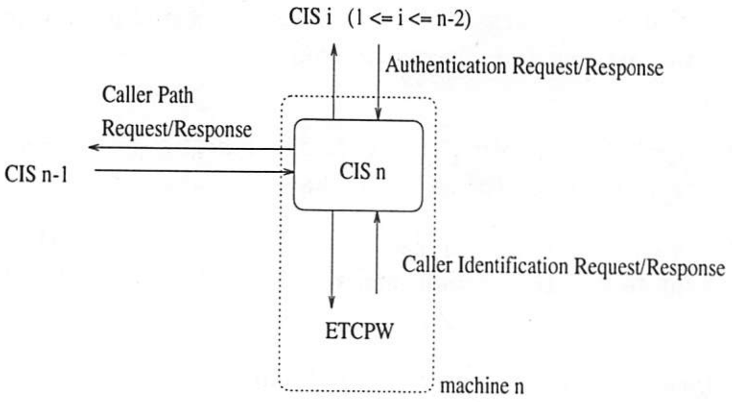


Figure 4: CIS's Requests/Responses

- *Caller Identification Request/Response Packet*:
  Caller Identification Request is sent by *ETCPW* to *CIS* to identify a caller. *CIS* will then respond to *ETCPW*, after checking out the caller's identity, by sending a Caller Identification Response Packet to *ETCPW*.
- *Caller Path Request/Response Packet*:
  When CIS_n sends a Caller Path Request to CIS_{n-1} , CIS_{n-1} responds by sending the complete information on the user's path (Caller Path Response) to CIS_n.
- *Authentication Request/Response Packet*:
  After receiving the path from CIS_{n-1}, CIS_n will send an Authentication Request to each of the machines in the path ($Machine_1$ to $Machine_{n-2}$). These machines will then send their Authentication Responses to CIS_n.

2.3.1 Caller Identification Request/Response

ETCPW sends a Caller Identification Request which includes the following parameters to the *CIS*.

```
(remote-IP-address, remote-port-number, local-port-number)
```

CIS then verifies the identity of the caller by checking with the *CIS* of other machines "mentioned" in the caller's path. This path along with a statement that tells the *ETCPW* whether the path was successfully authenticated, partially authenticated or not authenticated at all, is sent back to the *ETCPW* as a Caller Identification Response. If the statement sent by the *CIS* to the *ETCPW* says that the path has been successfully authenticated, *CIS* will record into its internal table the complete path of the user/caller. The information in this table is also referred to to form an appropriate response to the Caller Path Request or Authentication Request. Then depending on the Caller Identification Response, the *ETCPW* will decide whether to allow the requester access or not. Logging of the Caller Identification Request (sent by the *ETCPW* to the *CIS*) is done no matter what the outcome of the Caller Identification Response. This is done so that the logging information can later be used to help track a "hacker" in a post-mortem analysis.

2.3.2 Caller Path Request/Response

Caller Path Request is a request sent between the *CIS*s of the last two adjoining machines (i.e., $Machine_{n-1}$ and $Machine_n$). CIS_n of $Machine_n$ will first send a Caller Path Request (*TCP* port number) to CIS_{n-1} of $Machine_{n-1}$ in the form

```
(port-number of machine n, port-number of machine n-1)
```

CIS_{n-1} will then look up the process associated with the appropriate port number. Next it refers to its internal table for the complete network path of the user associated with the process. This complete path is then sent to CIS_n. That is, if a path exist from CIS_1 to CIS_{n-1} and the user-id of CIS_i ($1 \leq i \leq n-1$) is *user-id*$_i$, then the Caller Path Response will take the form

```
(user-id n-1, machine n-1:
 user-id n-2, machine n-2:
...
(user-id 1, machine 1)
```

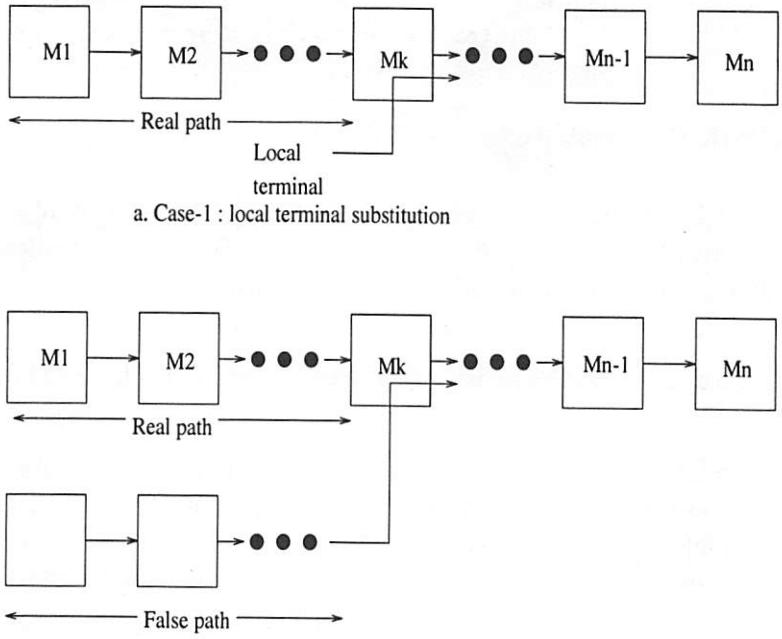
This Caller Path Request/Response packet and procedure is similar to that of the RFC1413. The main difference would be that the Caller Path Response of the RFC1413 only has information on the immediate requester machine and its user-id while our system's Caller Path Response has information on all machines prior to the destination machine and their user-ids.

2.3.3 Authentication Request/Response

The main aim of the Authentication Request/Response process is to verify the information received from the immediate requester machine (i.e., information on the Caller Path Response). The destination machine's *CIS* (i.e., CIS_n) sends, to the *CIS* of other machines "mentioned" in the Caller Path Response, Authentication Requests containing the parameter

```
(user-id)
```

The *CIS*s that received those Authentication Requests then checks their tables to see whether the appropriate user-id has a process. If so, they will send an Authentication Response of "yes", otherwise they will send "no"s.



a. Case-1 : local terminal substitution

b. Case-2 : False path substitution

Figure 5: False Path

3 Properties of Caller Identification System

When a network connection such as shown in Figure 3 has been established and all the *CIS*s in $Machine_1$ to $Machine_{n-1}$ have not been compromised, it is then possible to trace the network path from $Machine_1$ to $Machine_n$. If, however, one or more *CIS*s in the path (i.e. from $Machine_1$ to $Machine_{n-1}$) have been compromised, a "hacker" may lay a "false" path within these machines thus misleading other secure machines when they send Authentication Requests. Despite this, our Caller Identification System can identify at least one of these compromised machines. This ability to identify at least one compromised machine is an important property of our Caller Identification System.

Let us look at a situation where $Machine_k$ ($1 \leq k < n$) in a path from $Machine_1$ to $Machine_n$ has been compromised. The integrity of the information in $Machine_{k+1}$ to $Machine_n$ can be regarded as intact. That is, $Machine_k$ is assumed to be the last or the only machine within that path to have been "cracked". It is in this machine that the "hacker" can hide his/her original path. Therefore the path ($Machine_1$ to $Machine_{k-1}$) in $Machine_k$ is erased and a "false" path is installed to replace it. This "false" path may come in the two forms/cases:

- (case-1) : $Machine_k$ is the first machine (or home-base) of that false path
- (case-2) : A "false" path with a "false" home-base is laid down.

The Figure 5 diagrammatically depicts the above mentioned cases.

In case 1, the first machine after the "cracked" machine, $Machine_{k+1}$, will have the "false" path

$$Machine_k \Rightarrow Machine_{k+1}$$

The next machine, and the machines after that would all have the "false" path of

$$Machine_k \Rightarrow Machine_{k+1} \Rightarrow ... \Rightarrow Machine_i \ (k < i \leq n)$$

All these machines ($Machine_k$ to $Machine_n$), however, would still have in their "false" path, as the home-machine $Machine_k$. In other words, these machines may not have the real originating machine but it always retains enough information to detect at least one cracked machine.

Let us consider case 2, as shown in Figure 5b. In this situation, the "hacker" would create a "false" path and adjoin it to the "cracked" machine as his/her real path. If the next machine the "hacker" wishes to access is an uncompromised machine, then this machine will get some negative Authentication Responses from the *CIS*s of the machines in the "false" path. Therefore, $Machine_{k+1}$ will then be able to refuse access to itself, thus stopping the "hacker" from continuing his/her illegal activity.

A problem, however, arises when the "false" path given by the $Machine_k$ ("cracked" machine) to $Machine_{k+1}$ does exist (i.e., a path already established by another "innocent" user). In such a situation, CIS_i $(k+1 \leq i \leq n)$ will get positive Authentication Responses, to their Authentication Requests, from all the machines in the path even though that path is "false".

Nonetheless, the "false" path would contain information on the "cracked" machine ($Machine_k$). This information, unlike case 1, will be situated somewhere in the middle of the path and is not so easily detected. In such a case, the "hacker" has to be traced through manual means. That is, when the login information in the *CIS*'s tables is carefully analyzed, irregularities will appear and this will lead us to at least one compromised machine.

4 Conclusion and Future Works

In this paper, we have presented a system we designed called Caller Identification System. The purpose of this system is to strengthen the security in a local area network. The system consists of two main parts. Part 1 is the Extended TCP Wrapper (TCP Wrapper with extra features added to it) and part 2 is the Caller Identification Server (*CIS*). With the combination of these two parts we have a system that not only can identify unauthorized accesses but should hacking occur, can locate at least one compromised machine using post-mortem analysis. With this one compromised machine, we will eventually be able to track the hacker to his or her home machine. With this tool, we hope to discourage any potential hackers from trying to make an unauthorized access to a system and causing damage to it.

There are many useful features that can be added to the present design to make the system more secure. They include (1) the unassailability of the logging information in each machine (i.e., the logging information can never be manipulated, written over or wiped out) and (2) the security checks between *CIS*s during the authentication process.

Should any hacking occur despite our security measures, we will then require feature (1) (i.e., the unassailability of logging information) to trace the hacker down to his or her home-base. If the logging information can be wiped out or altered, the hacker could use this weakness to hide his trail. Logging information can be protected by:

1. storing a replicated logging information in another machine that is not known to users or

2. storing the logging information in an WORM (Write Once Read Many) storage

If a *CIS* responds to a Caller Path Request without verifying that the request is from another authentic *CIS*, then it is possible for a non *CIS* to request for information about any network path existing so far. This is something we must prevent. Verifying whether the Caller Path Request sender is a *CIS* or not can be done by using the public key method[3]. In the public key method, each *CIS* is assigned a public key (i.e., key that every other *CIS* knows). Thus, for CIS_{n-1} to send a Caller Path Response to CIS_n, CIS_{n-1} must first encode the Caller Path Response Message using CIS_n's public key. If the Caller Path Response sent by CIS_{n-1} is not received by CIS_n but by a "rogue program planted by a hacker in the same machine, the "rogue" program would receive a message that it cannot decode unless it has CIS_n's private key. There, however, exists one hole through which a hacker can access information on network paths. Should the hacker manage to obtain the private key of one of the *CIS*s, the Caller Identification System will not be able to stop network path information from falling into the hands of that hacker.

5 References

1. W. Venema, TCP Wrapper. Network Monitoring, access control, and booby traps. In *Proceedings of the USENIX Security III Symposium*, pages 85-92, September 1992.

2. M. St. Johns, Network Working Group - RFC1413, Obsoletes 931, Identification Protocol, February 1993.

3. R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM, 21(2), February 1978.

ATP
Anti Tampering Program

David Vincenzetti
vince@dsi.unimi.it

Massimo Cotrozzi
cotrozzi@dsi.unimi.it

Computer Science Department
University of Milan - Italy

ABSTRACT

Since the earliest days of the *Unix*$^{(tm)}$ operating system, it was clear that, being impossible for a system administrator to monitor every single change in the file system, it would be a fertile playground for intruders. Every time intruders get into a Unix system, they tend to install a backdoor, just in case the hole they used to get in is fixed. Usually this is accomplished by modifying a system executable to contain code that allows him/her to access the system without authentication. This action is commonly called "file tampering" or "leaving a Trojan Horse". The program we present has the target to discover whether the integrity of a system has been compromised; in brief, we have developed a tool called ATP (Anti Tampering Program) which performs post-attack analysis.

1. INTRODUCTION

In recent years, attacks to computer systems have grown in number and in sophistication: in the early days the most common method of intrusion was password "guessing", whereas lately there has been a radical change in techniques used, and now some network attacks don't even require the intruder to have an account on the machine. Almost every machine which is on the internet have been "visited", or at least, "knocked on" by an intruder. This kind of situation is not avoidable as it is just like having a door on the street: sooner or later somebody will try to see if it is open or not. Bad guys exist in the computer community as in the "real world". Not only do they exist, but, in the majority of the cases, they spend most of their time studying new techniques for intrusion. Studies have been made on the aims of the bad guys, and it was pointed out that a tiny percentage of "hackers" hack systems in order to destroy information or for espionage purposes. Most of the cases, in fact, involve people just acting out of curiosity, or just to make the computer community know they are better programmers than system administrators.

The Internet worm, in 1988, exposed a set of classic operating system bugs but today it is not only a matter of operating system bugs anymore. Malicious hackers are learning how to communicate secretly using cryptography, how to foul up common Unix daemons by changing the source address of TCP/IP packets and by enabling IP source routing, how to penetrate into a system which is "firewalled" by Cisco router access lists only. Furthermore Unix systems are configured, by default, in a very open way and most system administrators don't care much about enhancing their system's security level. Being connected to the Internet we are potentially vulnerable to some new attack techniques of which we are not aware, so we can hardly ever state that our system is absolutely safe. The eventuality of a successful attack to our system is a real thing; in such an event we must be equal to the situation. When a Unix system is successfully attacked by an intruder and he/she gains root privileges it's up to him/her to decide whether to delete all system files or "just" to install a backdoor. If the intruder deletes files or destroys data it will be extremely painful to repair damages, but, on the other hand, the act of vandalism would be surely noticed by the sysadm. Otherwise if he/she deletes nothing but installs a backdoor, probably nobody would realize it and the system's security would be entirely compromised from then on.

ATP was created to solve this problem. ATP is intended to give system administrators a tool to protect the system from unwanted modification of sensitive files, i.e. configuration files, system daemons, most frequently used binaries etc. These files are, in fact, the ideal candidates for backdoors.

2. ATP – ANTI TAMPERING PROGRAM

ATP "takes a snapshot" of the system, assuming that you are in a trusted configuration, and performs a number of checks to monitor changes that might have been made to files. Of course it is useless if system integrity has already been compromised. In this case one should reinstall system binaries from scratch and reconfigure the system in a safe way.

To perform file analysis, ATP creates a database of file attributes. For every file in the database some `stat` informations, a double checksum and some flags are mantained. `Stat` informations consist of UID, GID, permissions, datestamps, size, number of links etc. The double checksum is composed by two parts: a CRC32 checksum and a MD5 checksum. CRC32 (32 bit Cyclic Redundandy Check) is a very popular algorithm and it is actually used in a wide variety of situations; CRC32 is very fast and reliable in detecting errors. Unfortunately CRC32 isn't considered a *secure* hashing function since it wouldn't be too hard for a malicious hacker to tamper with a file and then add some weigthed padding to the file in order to make CRC32 match. That's why ATP uses MD5 in conjunction with CRC32. MD5 is a RSA-DSI product and it is considered by many the state of the art of signature algorithms. MD5 produces a 128 bits message digest and it is conjectured that the difficulty of coming up with any message having a given message digest is on the order of 2^{128} operations ($\approx 10^{38}$). CRC32 is fast while MD5 is likely to be overkill for most systems, so the first one is intended for frequent massive checkings while the latter should be used from time to time, for example on a nigthly basis. Speed is not a significant parameter in an environment where you monitor a small number of files, but it is an important consideration in an environment composed of several workstation, with many NFS-mounted file systems and a very large number of files to be monitored. After "profiling" ATP at execution time, we discovered that more than 95% of time used was for calculating checksums. That's why ATP doesn't really use the original MD5 but a *modified* version of MD5. The changes, suggested by its author Ronald Rivest, have been made in order to speed up the checks, but the algorithm, according to Rivest, is still capable of providing a good level of security.

The whole database is encrypted using the DES cipher. In particular DES is used in Cipher Block Chaining (CBC) encryption mode with a pseudorandom Initialization Variable (IV). The database itself is signed with MD5 prior to encryption. A database checksum, a revision number and a timestamp are maintained to assure that any tampering with the database file would be discovered.

3. THE DATABASE

When the database file (tipically `/usr/local/lib/atp_database`) is created, a database encryption key is requested and this key will be used, from here on, for every access to the database. An alternative method of specifying the key is to set the environment variable ATP_KEYWORD to the desired password (however we must point out that on some operating systems this would be very dangerous! For example on SunOs anyone can display the environment for all users with the command `ps -eauxwww`).

The database is first built by taking files as command line parameters. For every other invocation of the program, the files specified are added to the database, if they are not already present, or replaced otherwise.

As files are registered in the database filenames are converted to absolute, machine dependent pathnames. Filenames are worked out by following all symbolic links and NFS mount points.

The database is an ASCII file. Lines in the database are composed by the following fields:

- The actual host the file phisycally resides on
- The real pathname of the file on that machine
- I-node of the file
- File permissions
- Number of links

- User ID
- Group ID
- Size in bytes
- Modification time
- Creation time
- Double file checksum

The database name can be specified at compile time, by defining the variable DATABASE in the Makefile, or it can be specified on the command line via the " -f " option.

Four more actions can be performed upon the database:

- Printing on the standard output (" -c " option)
- Deleting files from the database (" -d <file>" option)
- Destroying the database (" -D " option)
- Manual editing the database (" -E " option)

The last action is somewhat dangerous, for it is the only moment when the database will be written down on disk in an unencrypted way, via the temporary files created by `vi` (which is the default editor). Two definable environment variables are related to this option: EDITOR and TMPDIR, which define the path in which temporary files are to be created. If a malicious hacker modifies the temporary file during a manual editing of the database, modifications would be successfully added to the database, because after any "hand modification", the checksum of the database is obviously recomputed before the final encryption. This option is thus intended for exceptional management only.

4. SETUID FILES

Setuid files are a problem in almost every computing environment, so a specific option have been provided to handle this situation.

Specifying the " -S " option on the command line, all SUID files are searched and then inserted into the database; the method for doing this is by using the `find` command which searches throughout the file system for all "sensitive" setuid files; the program can be told whether a setuid file is sensitive or not. As an example we consider sensitive a setuid file owned by root or any other system related UID (i.e. `bin`, `adm`, `sys`, etc.) to be sensitive, but we allow *joeuser* to have any setuid file owned by himself or by any other user. By editing `atp.config` it's possibile to set `find`'s search criteria and to create a list of sensitive UIDs.

5. CHECKS AND ACTIONS PERFORMED

There are two ways to perform checks on files stored in the database: a basic one and a custom one.

5.1. Basic Operation Mode

The checks which are executed in basic mode are obviously based upon the information stored for each of the files; the most significant tests are:

- Comparison of the checksum
- Comparison of the permissions
- Comparison of the UID and GID
- Comparison of the creation and modification times
- Comparison of the number of links
- Comparison of the file size

Three kind of actions can be performed as a consequence of the positive result of any of the checks. These actions can be selected on the command line using parameters " -k1, -k2, -k3 " which define the three levels of response to a file tampering. All three checking options monitor, for every file in the database, the consistency of the parameters stored with the effective parameters taken from the disk. The default checksum algorithm is MD5. Alternatively you can specify, via the " -C " option, the utilization of CRC32 only. This is for speeding up the checking process just in case you are continously running ATP in checking mode (i.e. by `cron` once an hour).
If the " -k1 " option was specified, a report is written on standard output for use by the security administrator. By specifying the " -k2 " option you force ATP to take the following actions upon failing some of the checks: "zap" the file (that is `chmod 000` and `chown root` the file) and notify the security administrator (and/or anyone specified in the ATP_RECIPIENTS environment variable) via `mail` and `syslog`.
The " -k3 " option, finally, adds to the actions performed by the two previous options the search for illegal SUID files, that is files which resides in the file system but which are not in the list of "legal" ones (which are already in the list). This is checked with the same command `/bin/find` as above (redefinable run time by means of the ATP_FIND_COMMAND environment variable).

5.2. Basic Mode Examples

The following example shows how the basic features of ATP can be used.

```
$ atp
atp:  Bad options.  'atp -h' for a kinda help
$ atp -h
```

```
atp -- Anti Tampering Program:  a file integrity checker for Unix
written by David Vincenzetti <vince@ghost.dsi.unimi.it>
Usage:  atp [options] [files...]
possible options are:
        <files>...        :  Add/replace files in the database
        -f <file>         :  Specify database file
        -E                :  Edit the database directly
        -c                :  Display database to stdout
        -d <file>         :  Delete file from database
        -D                :  Destroy the database
        -x                :  Change encryption key
        -t                :  Edit configuration file
        -S                :  Add SUID files into the database
        -C                :  Use CRC32 instead of MD5
        -k{1, 2, 3, 4}    :  Check for file tampering
        -k1      :  Plain check, send report to standard output only
        -k2      :  Like above + zap illegal files, use mail and syslog
        -k3      :  Like above + search illegal SUID files
        -k4      :  Enable custom checking

Relevant environment variables:
ATP_KEYWORD, ATP_EDITOR, ATP_FIND_COMMAND, ATP_RECIPIENTS,
ATP_MAILER, ATP_GIDLIMIT, ATP_UIDLIMIT
```

So, if this is the first time we run ATP upon our files, we would probably like inserting all SUID files in ATP's database.

```
$ atp -S
I see you have created no database yet:  creating a new one.

Setting database keyword.  Think it carefully!  Please don't choose
a short or obvious password since it may be vulnerable to
'guessing' attacks.  Please use at least a dozen characters.

Enter keyword:
Enter it again:

Scanning disk using:  /bin/find / -type f -print
inserted:  /bin/df
inserted:  /bin/newgrp
inserted:  /bin/passwd
inserted:  /bin/su
...
Database:  749 files; 77689 bytes
Database successfully saved.
```

Now we have control over all SUID files. Let's insert some more critical files in the database. As an example we select all "rc-files" and all files in the `/bin` directory.

```
$ atp /etc/*rc* /bin
Password:
VERIFIED database ‘‘/usr/local/lib/atp_dabatase’’ (77689 bytes)
         database MD5      :  b5e0046dd264aa70d5b72ef911ccb647
         last modification:  Aug 13 11:11:04 1993
         DES-CBC IV        :  40004c2840004c30

inserted:  /etc/auditrc
inserted:  /etc/bcheckrc
inserted:  /etc/brc
inserted:  /etc/d.cshrc
...
inserted:  /bin/ar
inserted:  /bin/as
inserted:  /bin/basename
...
replaced:  /bin/df
...
replaced:  /bin/newgrp
...
Database:  875 files; 90667 bytes
Database successfully saved.
```

Having inserted most of our critical files into ATP's database we can be pretty sure that any tampering will be detected. As an example let's suppose that an intruder illegally gained root privileges and he/she wants to install a backdoor on our system by changing the `/bin/login` program into a backdoor. This is what happens when we check things with the " -k1 " option:

```
$ atp -k1
Password:
VERIFIED database ‘‘/usr/local/lib/atp_dabatase’’ (90667 bytes)
         database MD5      :  b872f8772288b34a9a9da7bc7346f1e5
         last modification:  Aug 13 11:14:58 1993
         DES-CBC IV        :  40004c2840004c30

Comparing database with real files...

(1) FILE ‘ghost:/bin/login’:
SIZE has changed from 159744 to 171008.
MTIME has changed from May 12 11:05:16 1992 to Aug 13 11:25:40 1993
CTIME has changed from May 12 11:05:16 1992 to Aug 13 11:25:40 1993
```

```
MD5 had changed from e13c9975dc636effc3c14ff9e83fce1a to
0ee985c2eccd6681866569e70411edef.

WARNING! File tampering detected!
Please pay careful attention to the above files because they have
changed since the last database update.  Someone might have
deliberately tampered with them.  Please check it out.
```

5.3. Custom Operation Mode

ATP's configuration file (`atp.config`) permits extremely flexible file specification and management. ATP can powerfully handle files from a very complex and heterogeneous environment. The checks to be performed can be selectively defined according to file's importance. The "actions" which are to be taken when a certain file appears tampered are fully configurable.
The only way to edit `atp.config` is by using the " -t " option. With the " -t " option an internal, self contained editor is invoked; this is needed since the config file is stored to disk in enciphered form and what's more we want no plaintext temporary files to be created. The encryption algorithm and the integrity checks used for the database are applied to the config file as well.
ATP's configuration file has the following form:

```
regexp    check-list    action-list
```

`Regexp` defines a *File Domain* (FD), that is a group of files for which the checks defined in the `check-list` field are to be performed. Upon a failure of any of these checks the actions defined in the `action-list` field are executed. The `regexp` field describes ATP's internal pathname representation which is composed by two parts: a hostname and a standard Unix absolute pathname. When using the " -k4 " option the files listed in the database are examined. For each file the rules specified in the config file are applied. The checks are performed on a first match basis, like Cisco router access lists. So, it's adviseable to keep a default rule in the config file in order to guarantee that all files are checked.

Actions can be `#defined` at the beginning of the configuration file. This comes handy since you can create your own checking macros. However one builtin macro is provided, BLTN_ACTION, which acts like " -k2 " in the basic checking mode.

5.4. Custom Mode Examples

Suppose our config file looks like this:

```
#define SNAPSHOT "/usr/local/bin/make_snapshot >> /usr/adm/atp_log"
# Checking /bin/login for all hosts.  Let's check for Checksum,
# creation time, mod time, size, link(s) number and perms changes.
# In case of unexpected modifications perform builtin action and
# take a ''snapshot'' of the system.
*:/bin/login                Ccmslp   BLTN_ACTION;SNAPSHOT
# Let's check all /bin system files for ghost.
# In case of unexpected modifications take the standard steps
# for sysadm notification and file zapping.
ghost:/bin/*                Ccmslp   BLTN_ACTION
# Checking RPC services for hosts hp1, hp2 and hp4.
hp[124]:/usr/etc/rpc.*      Ccmslp   BLTN_ACTION
# Checking logfiles for ghost.
ghost:/usr/adm/*log         lp       BLTN_ACTION
# Default rule.
*:*                         Ccsp     BLTN_ACTION
```

The following example shows how to use the " -k4 " option:

```
$ atp -k4
Password:
VERIFIED database ''/usr/local/lib/atp_dabatase'' (13905 bytes)
        database MD5      :  40a3ed879d5b1dddbc4e23480165243b
        last modification :  Aug 13 11:27:48 1993
        DES-CBC IV        :  40004c2840004c30

VERIFIED config-file ''/usr/local/lib/atp.config'' (1539 bytes)
        config-file MD5   :  bdef1d3d85c6696d7fd0f68e17c7f7de
        last modification :  Aug 11 22:49:08 1993
        DES-CBC IV        :  8473847384738383

Comparing database with real files, applying config rules...

(1) FILE 'ghost:/bin/login':
SIZE has changed from 159744 to 171008.
MTIME has changed from May 12 11:05:16 1992 to Aug 13 11:25:40 1993
CTIME has changed from May 12 11:05:16 1992 to Aug 13 11:25:40 1993
MD5 had changed from e13c9975dc636effc3c14ff9e83fce1a to
0ee985c2eccd6681866569e70411edef.

Executing action:  '/usr/local/bin/make_snapshot...'
Action return code:  0

(2) FILE 'hp3:/bin/sh':
CTIME has changed from May 12 11:05:16 1992 to Aug 13 11:28:03 1993
```

```
PERMS have changed from -rwxr-xr-x to -rwxrwxrwx

WARNING! File tampering detected!
Please pay careful attention to the above files because they have
changed since last database update.  May be someone deliberately
tampered with them.  Please check it out.
```

6. CONCLUSIONS

ATP performs checks that can show with very high degree of confidence whether a file have been modified or not.

It is more powerful than other programs (such as Gene Kim's `Tripwire`) because of its sophisticated configuration language and its fully configurable set of constrains and actions for every single file, and it is more secure, for it is self contained and, with the exception of the " -E " option described above, it *never* writes to disk the content of the database in plain form. This is a very important point since the weakest part in Tripwire's security is the database integrity itself. In fact the first thing a malicious hacker does when he/she gains root privileges is to patch system binaries (such as `/bin/login`) and system daemons (such as `/usr/etc/in.remshd`) and alter Tripwire's database. It doesn't help much to protect the database by keeping it on a read-only mounted file system since the hacker can unmount it and mount it read-write. ATP doesn't require that the database be kept on a read-only media since the database cannot be meaningfully altered by anyone not knowing the right encryption key. Of course it would be better to backup ATP's database from time to time.

What's more, ATP provides a full mechanism for handling multi vendor NFS-mounted file systems (including HP-UX CDFs), and procedures that handle symbolic links through different filesystems. This is dramatically important in a distributed environment in which ATP can be executed from different clusters and files can have different names depending on the host we are accessing them from; in this case the program recognizes files which reside on a remote file system and if actions or checks are performed on such files, the same output is given.

Compared to ATP, Tripwire has two important drawbacks. The first is its incapability in following symbolic links. This is somehow surprising since files referred by symlinks are not registered in any way and this may lead to a false "sense of security". The second is the way it manages files from different hosts. Tripwire, in fact, keeps a different database for each system on which it runs. This works well when not using NFS but it becomes unacceptable otherwise. This is clearly shown in the following scenario. Suppose we have two systems, A and B, and we are using NFS for making A's `/usr` directory available to B. Alice, A's sysadm, thinks she'd better take care of, let's say, `/usr/etc/rpc.mountd`. So she inserts `/usr/etc/rpc.mountd` in Tripwire's database. Bob, B's sysadm, does the same. Then CERT announces and distributes a security patch about the mount daemon and Alice installs the patch on A. After installing the patch Alice updates Tripwire's database for host A. When Bob checks for the integrity of his system using B's database he will find out that `rpc.mountd` has changed and he will believe that someone has tampered with it!

7. FUTURE PLANS

In the near future we plan to design a method for handling largely distributed file systems: our effort will be in developing a sort of "atp server" called `atpd`, which should serve as a database centralization, to which requests should be directed for files which are not visible to the "local" file system. Of course all data transmission between master and clients would be encrypted; we are considering using public key encryption or some other method of cryptographic secret sharing.

8. AVAILABILITY

At present the program is in its final development state. The current version is 1.99. It's been tested on a variety of architectures such as HP, SunOS and SYSVR3. It will soon be available by anonymous ftp on `ghost.dsi.unimi.it` in the `/pub/security` directory.

For further info please contact David Vincenzetti, DSI – University of Milan, via Comelico 39, I-20135 Milan, Italy.

9. REFERENCES

[1] "Responding to computer security incidents: Guidlines for Incident Handling", Schultz, Brown, Longstaff University of California – 1990

[2] "Open Systems Security, an Architectural Framework", Arto Karila – 1991

[3] "RFC 1321: The MD5 Message-Digest Algorithm", R. Rivest, MIT Laboratory for Computer Science and RSA-DSI – 1992

The TAMU Security Package: An Ongoing Response to Internet Intruders in an Academic Environment

David R. Safford, Douglas Lee Schales, and David K. Hess
Supercomputer Center
Texas A&M University
College Station, TX 77843-3363

1. Abstract

Texas A&M University (TAMU) UNIX computers came under coordinated attack in August 1992 from an organized group of internet crackers. This package of security tools represents the results of over seven months of development and testing of the software currently being used to protect the estimated 12,000 networked devices at TAMU (of which roughly 5,000 are IP devices). This package includes three related sets of tools: "drawbridge," a powerful bridging filter package; "tiger," a set of easy to use yet thorough machine checking programs; and "netlog," a set of intrusion detection network monitoring programs.

2. Introduction

A Brief History of the Incidents

On Tuesday, 25 August 1992, the Texas A&M University Supercomputer Center (TAMUSC) was notified by the Ohio Supercomputer Center that a specific Texas A&M University (TAMU) machine was being used to attack one of their computers over the internet. The local machine turned out to be a Sun workstation in a faculty member's office. Unfortunately, this faculty member was out of town for a week, so rather than trying to gain access to the machine through the department head, it was decided to monitor network connections to the workstation and, if necessary, disconnect the machine from the net electronically. This decision to monitor the machine's sessions rather than immediately securing it turned out to be very fortunate, as this monitoring provided a wealth of information about the intruders and their methods.

The initial monitoring tools were very simple, but as the significance of what was occurring became apparent, the tools were rapidly improved to the point that the intruder's entire session could be watched in real time, keystroke by keystroke. This monitoring led to the discovery that several outside intruders were involved and that many other local machines had been compromised. One local machine had even been set up as a cracker bulletin board machine, which the crackers would use to contact each other and discuss techniques and progress!

By Thursday, 27 August, there was enough information about which machines had been compromised and how they had been broken into to allow an effective cleanup. In addition, the severity of the modifications the intruders were making, particularly on the bulletin board machine, made it imperative to stop the intrusions. The respective system managers, therefore, were contacted, arrangements made to shut down all machines, and a system cleanup scheduled for the next day.

On Friday, 28 August, the known affected machines were worked on, closing the security holes that had been used to break in, and all were brought back up on the network.

On Saturday, 29 August, an emergency call was received from one of the system managers, saying that the intruders had broken back into the cracker bulletin board machine. Concerned about the integrity of their research data, they asked for their machines to be physically disconnected from the rest of the network.

On Monday, 31 August, the logs of the new break-in were analyzed and it was determined that the crackers were much more sophisticated than originally believed and that many more local machines and user accounts had been compromised than initially realized. Several files were found containing hundreds of captured passwords, including ones on major (supposedly secure) servers. It appeared that there were actually two levels of crackers. The high level were the more sophisticated with a thorough knowledge of the technology; the low level were the "foot soldiers" who merely used the supplied cracking programs with little understanding of how they worked. Our initial response had been based on watching the latter, less capable crackers and was insufficient to handle the more sophisticated ones.

After much deliberation, it was decided that the only way to protect the computers on campus was to block certain key incoming network protocols, re-enabling them to local machines on a case by case basis, as each machine had been cleaned up and secured. The rationale was that if the crackers had access to even one unsecure local machine, it could be used as a base for further attacks, so it had to be assumed that all machines had been compromised, unless proven otherwise.

The recommendation to filter incoming traffic was presented to the Associate Provost for Computing on Monday afternoon and approved. The necessary equipment for the filter and monitor machines was bought or borrowed late that afternoon, and the design and coding of the filter proceeded through the night. Particular effort was made in the design to achieve the necessary security with the minimum of impact to local users. The filter was completed and installed by 5 PM Tuesday, 1 September.

At this point, the major task of analyzing all of the detailed logs and captured files was restarted. It was discovered that over 40MB of the cracker's tools had been captured, tools that they had FTP'ed onto some of the broken machines. These tools included Crack, network monitoring tools, all SunOS, Ultrix and Dynix source code (so they could replace any executable on the system), and cracking programs for virtually every CERT announced vulnerability. The logs showed that the crackers routinely placed back door and trojan login binaries on each broken system and used programs to set the timestamp and checksum of the replaced binaries to avoid detection.

On Thursday, 3 September, TAMUSC monitor logs showed an obviously automated attack by ftp that was sequentially probing every machine on campus. Here again it was decided to monitor this attack, as it was not clear what it could accomplish. This decision to observe, rather than immediately block, turned out to be very fortunate.

Shortly after midnight on Friday, 4 September, TAMUSC received a report from another site via the Computing Emergency Response Team (CERT) at Carnegie Mellon that the crackers had broken back into TAMU machines. The logs were immediately analyzed, and it was determined that the crackers had used ftp to install a program that allowed them to tunnel past the TAMU filter's blocks. In addition, even though they knew we were aware of their original intrusions, they continued their pattern of breaking in and replacing key system binaries.

At this point, the filter was completely redesigned to keep the crackers out, and the new version was installed by 5 AM Saturday. The new version changed the filter approach from "deny" based filtering (let everything in unless it is specifically denied) to "allow" based filtering (block everything unless it is specifically allowed). This new version, while providing much greater security, was unfortunately also more visible to valid users.

Since the new filter was installed, no successful intrusion attacks against TAMU machines have been observed, despite continued logging of probes and continued attempts. Recent efforts have centered in three areas: improving the ease of use and throughput of the filter, reducing the manpower requirements of the monitoring tools, and developing a program to help local system managers check their machines for proper security configuration.

Highlights of the Cracker Sessions

While all techniques used by the crackers aren't specified, the following section shows some of the more interesting things that were discovered. In all cases, references to specific machines have been changed; all of the spelling errors have been left in. The first fact is that the crackers seemed to have a compulsion to discuss their exploits with other crackers. While IRC seemed to be the preferred technique, many of the better crackers preferred less obvious methods, such as simply cat'ing directly to each other's ttys.

This snippet records an unnamed cracker on host1 talking with NMN (No Means No) concerning having run "ch", a brute force password cracking program on all 10 HP Snake machines in a Kent State lab. This conversation occurred on host1.tamu.edu using "cat >/dev/tty…".

NMN: "Well the people who run all 10 of the HP's will core when all thier logs show NMN.and especially if they find password crackers on them all.. me writing it is one thing ,but installing it is ANOTHER"

host1: "How would they find NMN on the hp.You lost me."

NMN: "Theyd see me Ftp to acct nmn on host2"

host1: "Possibly.Not liekly ..Unless they suspect something.Kent is the least concerened about security of any host i a have a ever been on.Oh maybe bsides .ai.mit.edu systems."

NMN: "Yeah ok"

host1: "Anyways.. dont forget to check up on the ch im running at host3.. its PID 16684 (ps -p 16684 will see if its still there) At host3, it should take 12.59712 days. *grin* BUT it sould crack it.. its running right now, it could crack it tonight, could crack it next week, who knows."

One extreme form of communication involved one cracker setting up a clandestine conversation bulletin board called LIMX (Local Immediate Message eXchange) on host4.tamu.edu, a machine that they had broken. LIMX was implemented as a passwordless back-door login by replacing the login executable. Here is the logout message from LIMX:

> "Connect to us again sometime, dont forget to spread the word about our system (host4.tamu.edu/128.194.xx.xx) and the account name (limx) to all your friends. If you dont have any friends, thats ok, tell your family members! Imagine the fun at the dinner table if Mom, Dad, sister and brother (and of course your dog fido) all had computer terminals and were connected online to the BACKDOOR LIMX, all chatting about the latest gossip, sports reports, fashon tips, and the shocking crimes being committed (which of course aren't related to our wonderful hypocrisy 'democracy'), and of course the wonderful meal moms been slaving all day over a hot kitchen microwave making, which none of you can comprehend. So spread the word! And connect to us again! Anonymous Cyberpunk Number One Oh yeah, and thanks to NoMeansNo for making this wonderful program! Sure beats IRC!"

3. Package Overview

Response Overview

Response to the intrusion incidents has three major thrusts: filtering, monitoring, and cleaning. The first line of defense is the bridging filter package *drawbridge*, which is used to filter all packets to or from the internet. *Drawbridge* allows internet access to be controlled on a machine by machine and port by port basis on a full T1 bandwidth basis. Using the filtering built into the TAMU WAN router (cisco) was initially considered, but it was determined that our requirements, particularly in the need for supporting potentially different filtering to each of the roughly 5,000 IP machines in the TAMU class B network, were too complex for the router. In addition, something was needed that could handle full T1 bandwidth, was itself very secure, and could be implemented rapidly. While other firewall configurations are known to be stronger, *drawbridge* provides a level of compromise between security and availability more acceptable to the university environment and provides much needed flexibility and throughput for the TAMU large scale network.

Realizing that *drawbridge* was a compromise between convenience and security, a set of monitoring tools was developed to look for intrusions that might be attempting to circumvent the filter. These tools continuously monitor the internet link, checking for unusual connections, patterns of connections, and for a wide range of specific intrusion signatures.

The third major thrust has been the development of the *tiger scripts*, an automated tool for checking a given machine for signs of intrusion and for network security related configuration items.

Figure 1 shows an overview of the filter and monitor implementation. In traditional secure gateways, a filter and secure bastion host are used and all traffic to or from internet is forced through them. This typically means that users need proxy clients for external access, such as for telnet and ftp, so that they all do not have to log on to the bastion host for external access. At TAMU, the filter allows arbitrary protocol filtering on a host by host basis, so that each department can set up its own authorized hosts with their own service configurations (subject to the campus wide minimum standards). This provides a reasonable level of both security and flexibility for educational and research requirements. For a UNIX host to be enabled at all beyond the default incoming permissions for mail, it must pass the *tiger scripts*, as described later. The monitor node is placed outside the filter so that it can record connection attempts which are blocked by the filter. This placement has been crucial to recognizing intrusion attacks, but does place the monitor itself at risk. To minimize this risk, both the filter and monitor are placed in a controlled access machine room and the monitor is configured for secure network access. The filter is similarly programmed only to respond to secure filter update requests, which are not routeable.

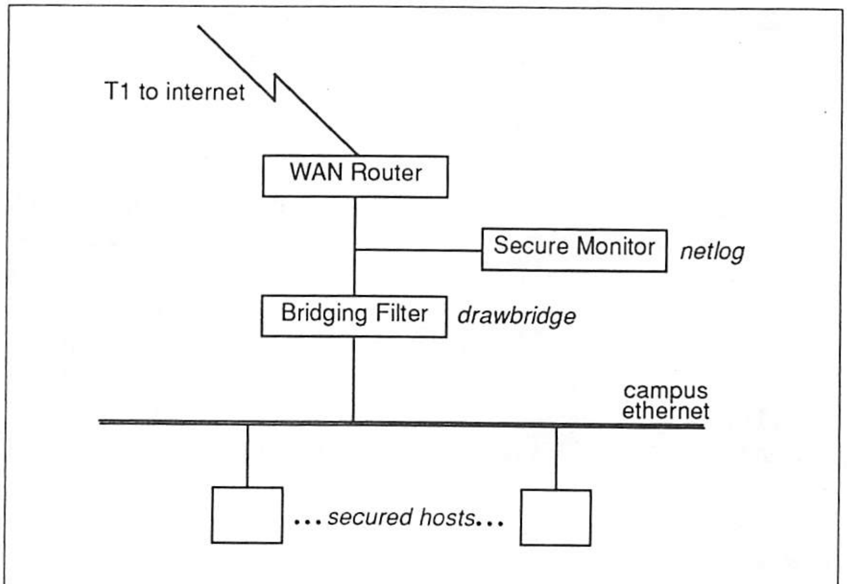


Figure 1. Overview of Components.

Filter (drawbridge)

Chapman [1] presented an interesting analysis of the limitations of current filter implementations at the Third UNIX Security Symposium. The drawbridge program, along with its support filter specification language and compiler, address some of his critical recommendations with respect to both functionality and ease of specification.

The filter approach chosen was based on a PC with two SMC 8013 (AKA Western Digital) ethernet cards. The first software implementation was based on pcbridge by Vance Morrison. This initial version was soon rewritten from scratch in C (compiled with turbo C++) to make the addition of needed features somewhat easier. The current filter design provides "allow" based filtering per host with separate incoming and outgoing permissions.

For both performance and configuration management, the filter tables are created on a support workstation, based on a powerful filter configuration language, and then securely transferred to the filter machine, either at boot time or dynamically during operation. The support machine does all the hard work of parsing the configuration file, looking up addresses, and building the tables, so that the filter itself need only perform simple O(1) table lookups at run time. Updating the tables dynamically is made secure with a Data Encryption Standard (DES) authentication.

The current default configuration allows any outgoing connection, but basically allows in only smtp (mail). Several campus and departmental servers have been checked and set up as hosts that are able to receive incoming telnet, ftp, nntp, and gopher requests.

Monitor

The goal of monitoring is to record security related network events by which intrusion attempts can be detected and tracked, particularly in those services allowed through the filter. This is a very difficult problem in general. The communication data rates make this problem somewhat like trying to take a sip of water from a fire hose; TAMU has some 30 terabytes of internal data transfer per day, and its internet connection is on the order of 4 gigabytes per day, with an average of 100,000 individual connections during that period. Clearly, monitoring needs to be both very selective and flexible, and automated tools are needed for reviewing even these resultant logs. Another problem is that of monitor placement. It is important that monitors be placed so that critical segments can be observed and so that the monitors themselves are secure.

Our solution includes the programs *tcplogger*, *udplogger*, *etherscan*, *nstat*, and some associated support programs. The tcp and udp loggers basically log a one line summary for all connection attempts. The associated analysis programs report on suspicious connections or patterns of connections. In addition, these logs have been very useful in analyzing details of security events after the fact. The *etherscan* program goes much further, actually scanning all packets and their contents, looking for a specific set of intrusion signatures, such as root login attempts from off campus. The *nstat* program collects statistics on all traffic to the filter and is useful both for capacity planning and for detecting unusual activity patterns. *Nstat* detected a clandestine FSP server on campus that was providing a repository of pirated commercial software, simply by noting a large transfer rate on a specific UDP port.

Machine Cleanup (the tiger scripts)

The phrase 'Tiger Scripts' comes from the concept of a 'tiger team.' A tiger team is a group which locates problems in a security system and demonstrates this problem by using it to circumvent the security system. By doing this, it is hoped that any weakness will located and corrected. The 'Tiger Scripts' perform the first part of this task in the UNIX environment. They search through a UNIX system and report any elements of it which may represent a security risk.

After a series of intrusions were discovered at Texas A&M University, it became apparent that a large and unknown number of machines attached to the campus network had been compromised. A filtering bridge was installed between the campus network and the Internet in order to protect the machines at the campus. It was still necessary though to clean up the machines that were involved. Most (if not all) of these machines were UNIX systems, but there were only a few people available at the university with the knowledge to locate and correct the problems on these machines. The 'Tiger Scripts' were developed to search out and report these types of problems. The scripts are also used as a means of verifying the security of machines to which access is allowed through the filtering bridge. Because of continuing development, the scripts have grown beyond just this internal use.

There were several goals for the 'Tiger Scripts.'

- Ease of use
- Robustness
- Portability
- Functionality

It was essential that the scripts be easy to use, as they were to be used by persons who possibly had little UNIX systems management background. Towards this end, no user configuration of the scripts are required. Once unpacked, all that is necessary is to run the script 'tiger.' This generates a report which contains any possible problems found. All messages are tagged with severity levels. These severity levels are used to indicate whether the system would be cleared through the filtering bridge. At the university, the manager of a system simply provides a copy of the report when requesting access.

It was also essential that these scripts be robust. Since they would be run on many different systems, the scripts had to be written in order to handle the nuances of the many administrators. This is of most concern when parsing system configuration files, as this is often where security problems manifest themselves.

In addition to these goals, portability had to be considered as well. The flavors of UNIX running on machines at Texas A&M is diverse. The initial release of the scripts (October 1992) was targeted toward machines running SunOS 4.1.x, as these machines were the primary target during the intrusion. The second release (April 1993) incorporated support for SunOS 5.x and NeXTOS 3.0. Future releases will include support for other UNIX derivatives such as AIX 3.x, HP-UX, IRIX, and UNICOS. The scripts are designed, however, to perform a "best of their ability" attempt even when specific support is not provided for a system. The only primary requirement is that the systems Bourne shell support the defining of shell functions.

The scripts accomplish these goals, while at the same time providing the following functionality. System configuration files are checked for problems, system binaries are checked for alterations, or known security problems, and known signs of an intrusion are checked. There is also support for the extension of the checks as new problems are reported.

For ease of use, the *tiger scripts* label all outputs with an error classification:

ALERT	A positive sign of intrusion was detected.
FAIL	The problem that was found was extremely serious.
WARN	The problem that was found may be serious, but will require human inspection.
INFO	A possible problem was found, or a change in configuration is being suggested.
ERROR	A test was not able to be performed for some reason.

As an aid to ease of use, an explanation facility is provided with the *tiger scripts*. Explanations can be requested for specific messages, or an explanation report can be generated for the security report. The explanation report can automatically be inserted into the security report, with explanations following each of the security messages. The explanations describe the situation, why it is a problem, and how to correct it. Figure 2 shows an excerpt from a security report with explanations inserted.

The checking performed covers a wide range of items, including items identified in CERT announcements and items observed in the recent intrusions. The scripts use cryptographic checksum programs to check for both modified system binaries (possible trap doors/ trojans), as well as for the presence of critical security related patches.

At the present time, the *tiger scripts* have been configured for SunOS 4.1.x, SunOS 5.x, Nextstep 3.0, AIX 3.2, HP-UX, IRIX 4.0, and UNICOS 7.0 releases. The programs are largely table driven for ease of porting, and ports to other platforms are being worked on.

```
--WARN-- [acc012w] Login ID sundiag has uid == 0.

The listed login ID has a user ID of zero (0) and is not the 'root' account.
This should be checked to see if it is legitimate.  In any case, having login
ID's with a user ID of zero tends to lead to security problems, and should be
avoided (except for 'root')

--WARN-- [acc004w] Login ID csehlhp is disabled, but has a .rhosts file

The listed login ID is disabled in some manner ('*' in passwd field, etc), but
a non-zero length .rhosts file.  This can allow the login ID to continue to be
used.  Unless this has been specifically set up to provide some service, it
should be removed.

 --INFO-- [acc002i] Login ID vu18368 is disabled, and has a shell of /bin/login.

The listed login ID is disabled, but has a potentially valid shell. These can
usually be safely ignored, but should be checked.
```

Figure 2. Sample *tiger scripts* security report with explanations inserted.

Policies

The policies and procedures need to provide both security and flexibility. The resultant decision was to filter incoming traffic other than mail to all machines and then allow case by case requests for authorized hosts status, based on successful demonstration of basic security configuration with the *tiger scripts*. Special requests for allowing incoming requests to special servers that are not easily checked, such as for embedded robot controllers, have been made. In these cases, the connections have been allowed, but special monitors have been implemented on these services.

Long term policy questions that remain unanswered include how to handle updates in response to critical CERT announcements and how to handle OS updates. Obviously, some way to coordinate both periodic and quick response host reviews is needed. This filter configuration language does support machine classes, so it would be possible to do something such as "disable ftp to all SunOS 4.1.1 machines" in response to a CERT announcement of a respective problem, but it would be nice to have a mechanism to communicate such announcements to the respective managers *before* cutting off access. The problem on a large campus is maintaining a contact list for a large number of machines, given the high rate of turnover in student managers. In addition, the information in the filter configuration file may rapidly become outdated, as managers update their machines' hardware and software. The current plan is to require periodic (annual) security checks with the current *tiger scripts*, enforced with the possible loss of IP authorization. In the case of aperiodic security events or announcements, an attempt will be made to evaluate the time criticality of responding and require appropriate event specific checking. As the *tiger scripts* are easy to run, it is anticipated that this requirement will not be a significant burden to system managers.

A recent case in point was the announced security problem with the wuarchive anonymous ftp code. In this case, it was known exactly which machines had ftp authorized, and the respective managers were contacted immediately. The managers updated their software so rapidly that it was not necessary to block access, and the limited number of authorized machines avoided the need for an immediate *tiger* update.

4. Filter (drawbridge)

Design

Drawbridge is different from any of the current standard firewall configurations. Using the categorization of firewalls developed by Ranum [2], *drawbridge* compares best to a filtering router firewall configuration

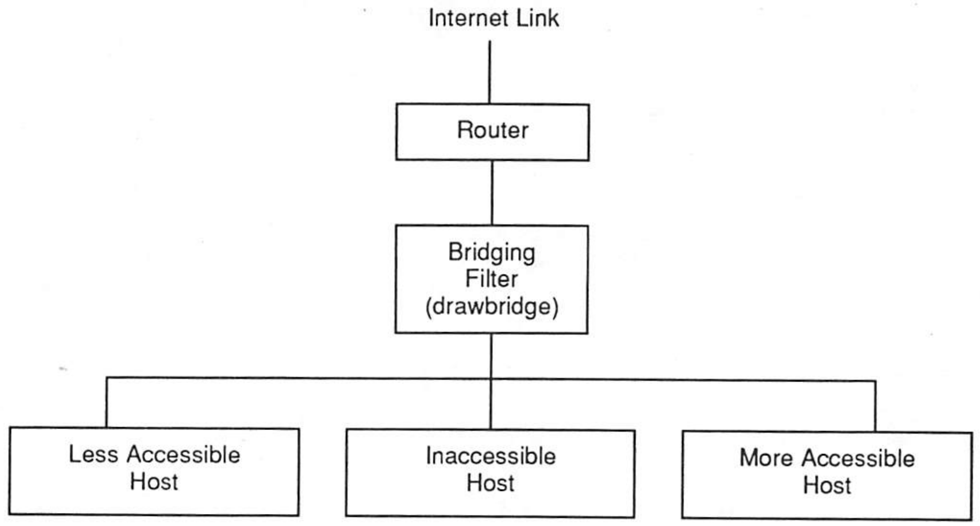


Figure 4. *Drawbridge* firewall.

Rather than repeat that material, it will be assumed that the reader is familiar with packet filtering and a discussion of how *drawbridge* tries to address some of these problems mentioned by Chapman [1] will be presented.

One of the first problems with current packet filtering implementations is that they are difficult to configure. They use a simple syntax that is designed for efficient implementation, not for effective configuration by an administrator. On a university campus, there is a need for many different filtering configurations to satisfy the diverse needs of the many users. Also, while the needs of administrators are usually defined in terms of connections, filters usually are defined in terms of packets only; the semantics of connections must be tediously mapped on to them.

These issues are addressed in *drawbridge* through the use of compiled tables. One table is defined for each (entire) IP network with each host address in that network being a single entry in the table. This allows a powerful source language to be designed that administrators can easily use and that is flexible enough to define complex sets of filters. In addition, *drawbridge*, under TCP, is not restricted to filtering in terms of packets but also filters in terms of connections. This makes configuration easier for the administrator and *drawbridge* more efficient.

This table design allows arbitrarily complex filters to be defined with little penalty. In conventional filtering routers, as filters are added, the performance begins to quickly drop due to how they implement the filtering rules. In *drawbridge*, arbitrary numbers of complex filters can be set up and the performance remains almost constant since simple look ups are performed and only connection establishment packets are filtered for TCP.

A second problem with most filtering implementations is that testing filter configurations is difficult. *Drawbridge* remedies this by allowing the administrator to check the results of a compiled configuration file to see if the correct filtering rules have been applied. Since *drawbridge* is less algorithmic than current filtering implementations, it is sufficient to investigate the compiler output. The administrator can look at the class that a host has been assigned and at the filtering lists defined for each subtable in that class.

A last problem that *drawbridge* addresses is the need for support for source port filtering. *Drawbridge* specifically defines an entire subtable to support TCP source port filtering (UDP source port is not currently supported). Since source port filtering does allow the possibility of tunneling, *draw-*

as shown in figure 3. In a filtering router firewall, a router which has packet filtering support is used to filter packets to and from hosts on the "inside" of the router. This is used to establish a policy where hosts are provided more or less access depending on the decisions of the network managers.

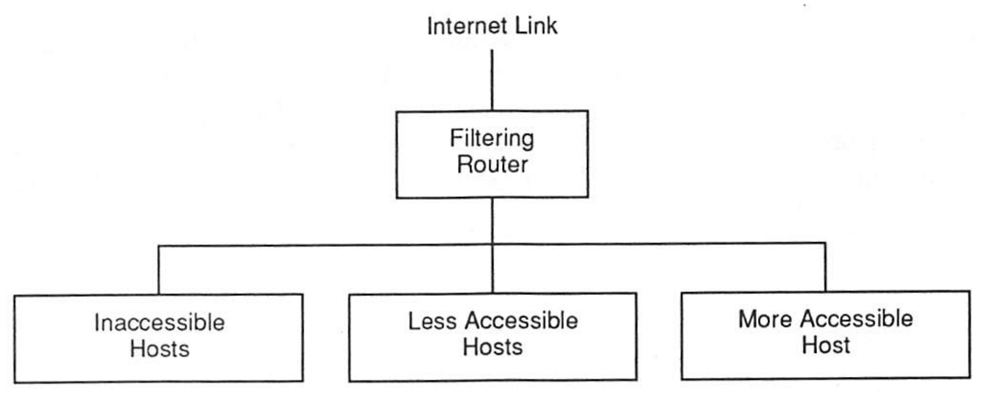


Figure 3. Typical filtering router firewall.

In an university environment, access typically would be based on the department the machine is located in, who manages/uses the machine, and an initial security audit, which is hopefully repeated on a regular basis. At TAMU, security audits are performed using the *tiger scripts* package, which was developed for this purpose. Any hosts that are never "registered" for access through the filter would receive some type of default access that would be defined by the network managers.

While this type of firewall is theoretically weak in comparison to other firewall methods, in practical use it does provide a useful increase in security. The points of attack have been greatly reduced and casual intruders are quickly discouraged.

A typical *drawbridge* firewall configuration is related to a filtering router firewall as shown in figure 4. The difference is that instead of using a filtering router as the firewall, the filtering function is moved from the router into *drawbridge* which acts as a bridging filter. Note, however, that figure 3 describes just a typical setup; a router is not a necessary component of a *drawbridge* configuration.

Comparison to Other Filtering Methods

Chapman [1] is an excellent source of information about packet filtering issues. He discusses the concepts behind packet filtering and some of the problems associated with it. He also discusses the problems with current implementations of packet filtering found in some current routing products.

Some of these problems include:

- Complex configuration language
- Difficult verification
- Lack of filtering on key parameters, such as source port or direction

bridge does add the restriction that the destination port must be greater than 900; 900 was chosen due to certain FTP implementations that happen to use FTP data ports beginning at around TCP port 900 rather than following the BSD convention of starting at 1,024.

Physical Structure

Drawbridge is physically structured as shown in figure 5. The PC running *filter* is placed between the external network (Internet link) and the internal network (campus) that will be protected. Optionally, a Sun workstation can be used to communicate with and manage *filter* on the PC. *Filter* acts as a filtering bridge between the external and internal networks. *Filter* performs bridging but does not conform to bridging standards, e.g., it has no support for Spanning Tree Protocol.

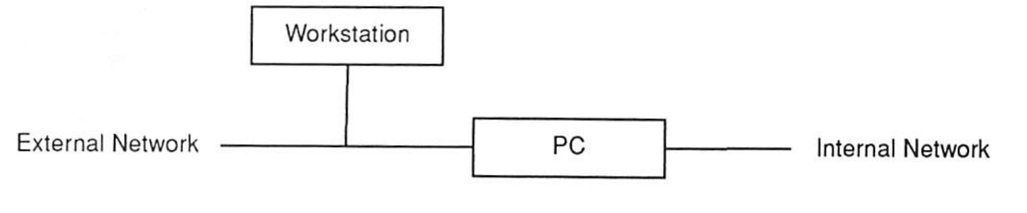


Figure 5. Ideal *drawbridge* Configuration.

In the ideal configuration, the workstation would be placed outside of *filter* so that monitoring of connection attempts from the external network can be performed. This is a good way to look for attacks and probes that are attempted against your internal network (and are hopefully blocked by *filter*). Since this also restricts the workstation's access to the internal network, a workstation will have to be committed specifically for this purpose. If a spare workstation isn't available, a machine on the internal network can be used to perform the management.

As mentioned above, *filter* is a table based filtering bridge. This approach was taken to improve the performance of filtering. The tables are generated by the following process (see figure 6). First, a source file containing filtering specifications in a special language is generated and maintained by an administrator. This file is then passed through *fc*, which generates the tables used by *filter*. These tables can be loaded via *fm* or by floppy disk.

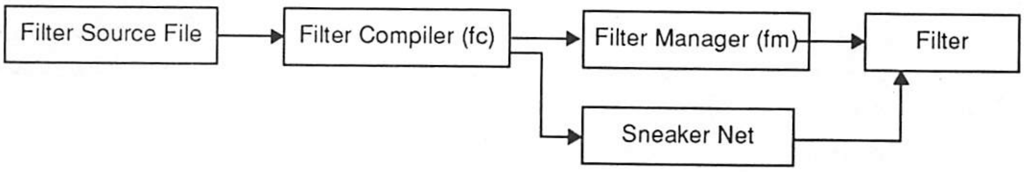


Figure 6. *Drawbridge* Compilation and Loading Process.

Filter Compiler Language

The language used by the compiler contains constructs for creating the various tables used by the filter. Constructs exist for specifying the network access on a per host basis, on a network or on a subnetwork basis. Groups of services can be created. These groups can be used in cases of related services or to group related machines. Access to particular external sites can also be granted, and access from certain sites can be denied. These constructs are shown in figure 7.

```
host (<hostname>|<ip_address>) <list of service_entries>
network (<ip_address>|<ip_address> - <ip_address>) <netmask> <list of service_entries>
define <group_name> <list of service_entries>
allow <ip_address> <netmask> <list of service_entries>
reject <ip_address> <netmask>
```

Figure 7. Constructs contained in the *filter* compiler language.

Hosts and networks can be granted network access using service specifications or group names. As hosts and networks are processed, the classes used by the filter are created. Hosts with equivalent network access (real access, not syntactic) will belong to the same class.

A group is a list of comma separated service specifications or other previously defined groups. Groups can be used to relate services or to categorize machines, allowing quick global changes to a category of machines. The special group "default" specifies the default access for any machine that does not match any of the networks loaded into *filter*.

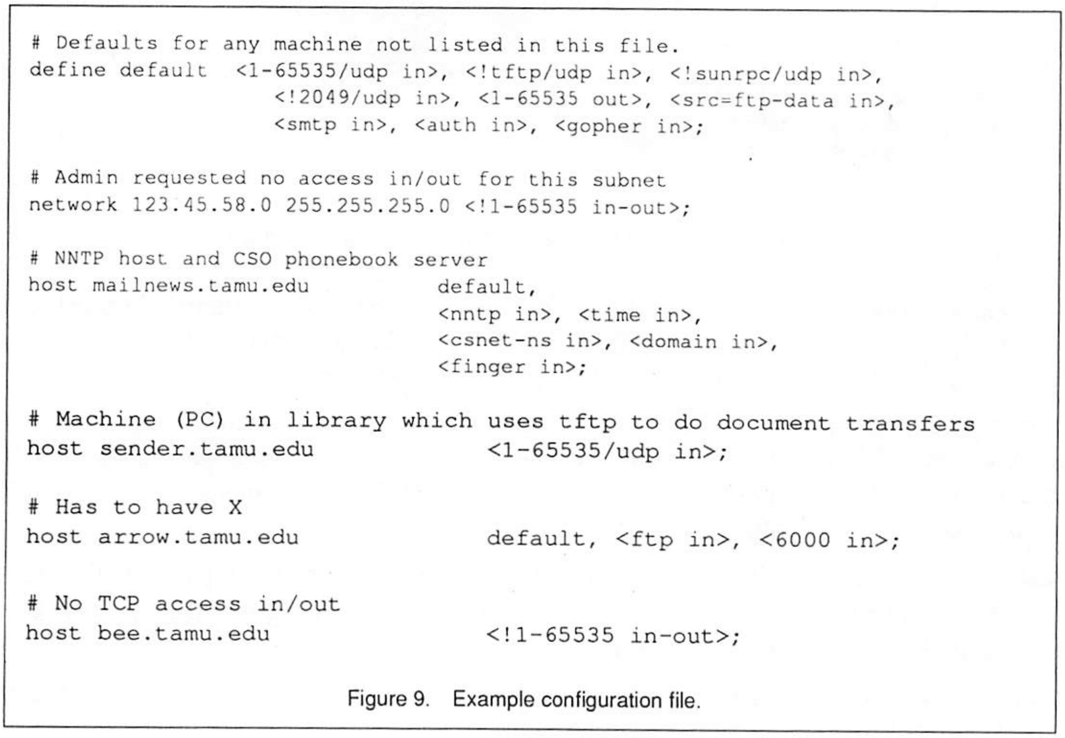
Constructs also exist for building the allow and reject tables used by the filter. The allow table allows internal machines access to a restricted external service. The reject table is used to block all incoming packets from a host or network.

The basic element of the language is a service specification. The service specification contains four pieces of information: the service, protocol, source or destination, and traffic direction. The service can be either an entry from /etc/services or a numeric port. Service ranges can also be used. The protocol specifies the protocol the service uses. The source or destination indicates whether the filter should use the source port or the destination port. Finally, the traffic direction indicates whether this is for outbound packets, inbound packets, or both. The grammar defining a service specification is shown in figure 8.

An example configuration file is shown in figure 9.

```
<service_entry>   ::= <  (src=|dst=) <service_desc> (in|out|inout) > |
                      <! (src=|dst=) <service_desc> (in|out|inout) > |
                      <group_name>
<service_desc>    ::= <service> | <service_range>
<service>         ::= <port_number>               |
                      <port_number>  / <protocol> |
                      <service_name>              |
                      <service_name> / <protocol>
<service_range>   ::= <port_number> - <port_number>                |
                      <port_number> - <port_number> / <protocol>
<protocol>        ::= <protocol_number> | <protocol_name>
```

Figure 8. Service entry grammar.



```
# Defaults for any machine not listed in this file.
define default  <1-65535/udp in>, <!tftp/udp in>, <!sunrpc/udp in>,
                <!2049/udp in>, <1-65535 out>, <src=ftp-data in>,
                <smtp in>, <auth in>, <gopher in>;

# Admin requested no access in/out for this subnet
network 123.45.58.0 255.255.255.0 <!1-65535 in-out>;

# NNTP host and CSO phonebook server
host mailnews.tamu.edu          default,
                                <nntp in>, <time in>,
                                <csnet-ns in>, <domain in>,
                                <finger in>;

# Machine (PC) in library which uses tftp to do document transfers
host sender.tamu.edu            <1-65535/udp in>;

# Has to have X
host arrow.tamu.edu             default, <ftp in>, <6000 in>;

# No TCP access in/out
host bee.tamu.edu               <!1-65535 in-out>;
```

Figure 9. Example configuration file.

How filter Works

fc generates four different kinds of tables (see figure 10):

- Multiple "network" tables, with one entry per host address,
- A "class" table with one permission list per distinct service class,
- A global "allow" table, and
- A global "reject" table.

The network tables have an entry for each host in the network. The host portion of an address (ignoring any subnetting) determines the index into the table. The value in the table defines the "class" that will be applied to a host when a packet is to be filtered. Only class B and C networks are currently supported as *filter* does not have the capability to use any memory above 1 MB.

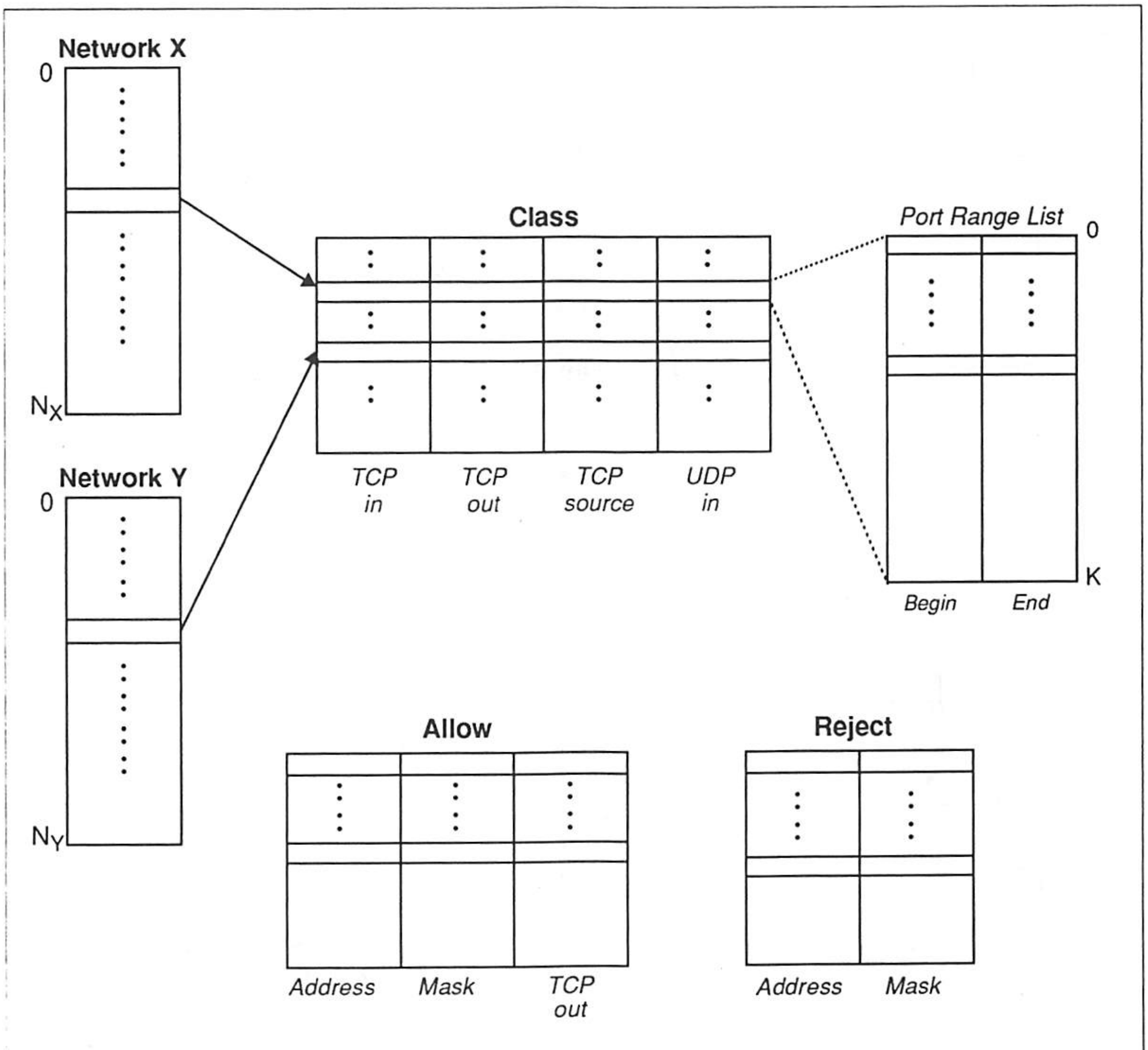


Figure 10. The four tables generated by *fc*.

Note that the network and class tables are defined in terms of a host on the internal network. No filtering is done based on the address of a host outside of the filter except on a global basis for reject and allow. It is assumed that an inside host will control which outside hosts are allowed to access its services, e.g., using TCPWrapper. *Filter* only controls which internal host's services are open, not which external hosts may access an internal host's services.

The host's class is used as an index into the class table. This second table is composed of four subtables: TCP in, TCP out, TCP source and UDP in. The subtables are composed of lists that contain port number ranges. A class specifies a list out of each subtable that defines a host's filtering. It is important to note that the TCP filtering only occurs when ACKless SYNs (connection initiation) are detected in a TCP header. All other packets of a TCP session are not filtered. Also, all UDP packets are filtered on an incoming basis only.

The last two tables are the allow and reject tables. The allow table globally allows packets out from any machine on the inside of the filter to the list of addresses in the allow table using the supplied list of port number ranges. The reject table globally rejects packets coming in from any machine on the outside of the filter with an address corresponding to an address in the reject table.

Figures 11 and 12 are flowcharts describing how *filter* uses these tables to check connections for incoming and outgoing requests, respectively.

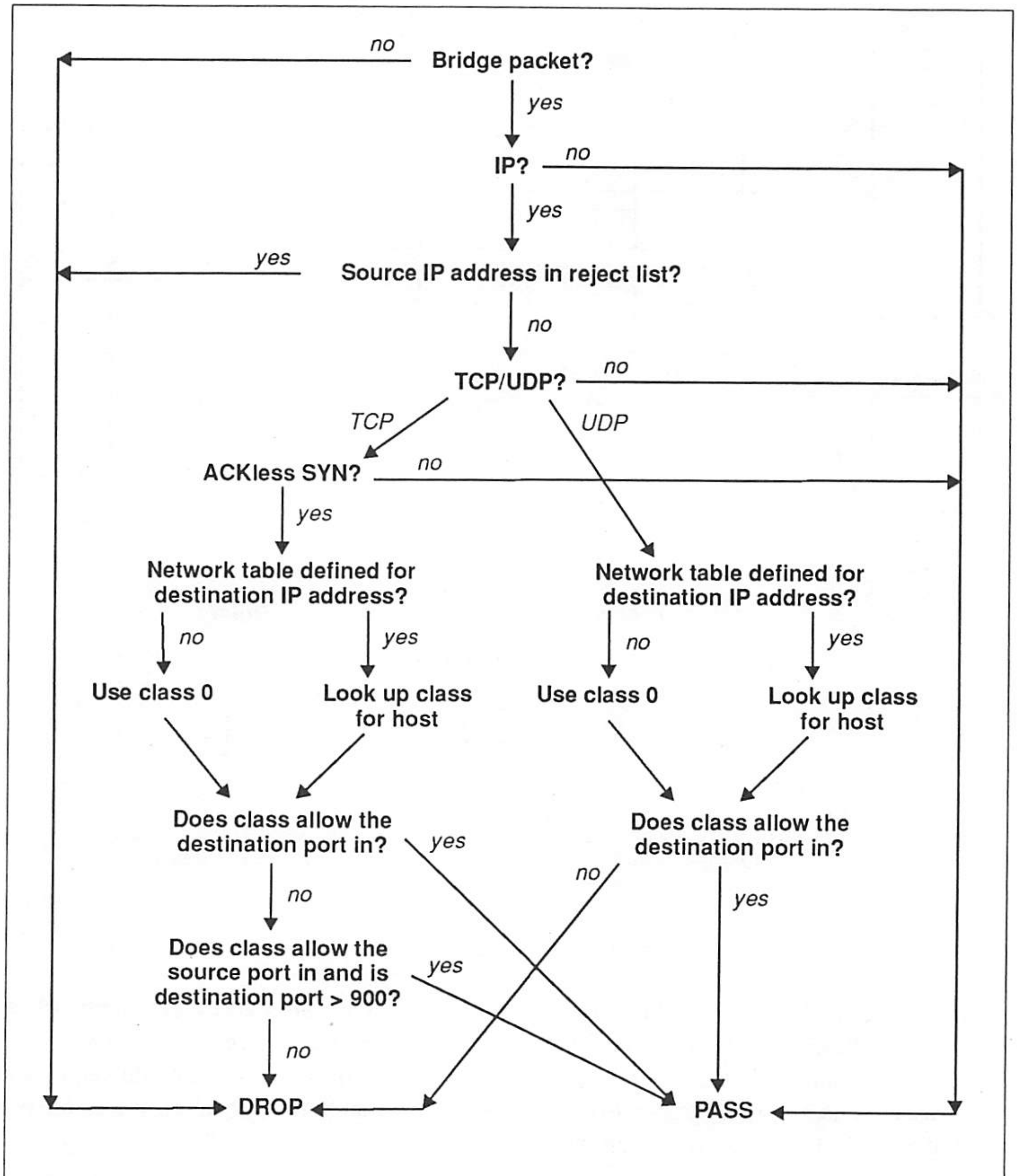

Bridge packet?
no
yes
IP?
no
yes
Source IP address in reject list?
yes
no
TCP/UDP?
no
TCP
UDP
ACKless SYN?
no
yes
Network table defined for destination IP address?
no
yes
Use class 0
Look up class for host
Network table defined for destination IP address?
no
yes
Use class 0
Look up class for host
Does class allow the destination port in?
yes
no
Does class allow the source port in and is destination port > 900?
yes
no
Does class allow the destination port in?
no
yes
DROP
PASS

Figure 11. Incoming Packet Filtering Algorithm.

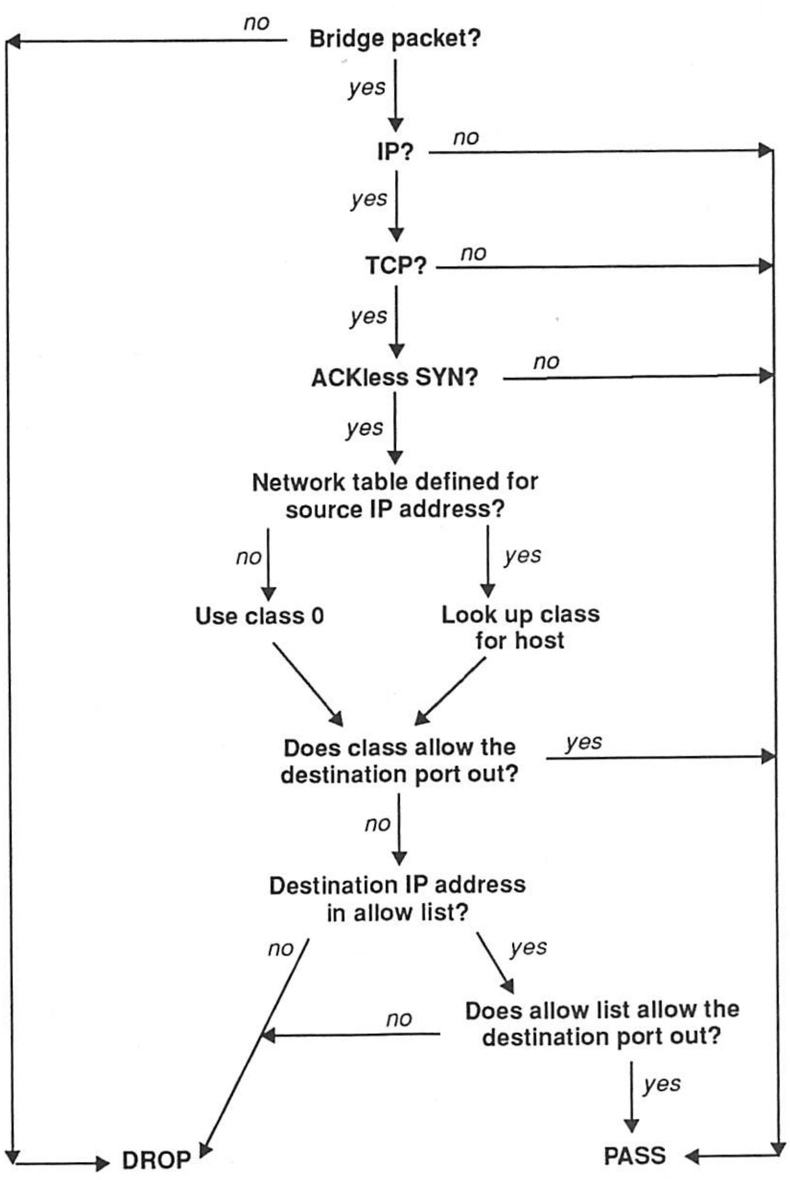

no
Bridge packet?
yes
IP?
no
yes
TCP?
no
yes
ACKless SYN?
no
yes
Network table defined for source IP address?
no
yes
Use class 0
Look up class for host
Does class allow the destination port out?
yes
no
Destination IP address in allow list?
no
yes
Does allow list allow the destination port out?
no
yes
DROP
PASS

Figure 12. Outgoing Packet Filtering Algorithm.

Implementation

Filter started out as a simple modification to PCBridge, a public domain program written in assembly. It is now over 3,000 lines of C code. However, the efficiency of PCBridge has been retained and even improved upon. *Filter* double buffers outgoing packets and has code to perform adaptive bridging using a hash table.

Figure 13 shows the performance of *filter* under different loads. The input packet rates were generated and monitored with a combination of two workstations and an ethernet analyzer with the packets generated on one side of the filter and measured on the other. These packets were built such that all of them would pass through the filter (not be bridged) and would not experience any higher level protocol backoffs. A set of fixed sizes and a sample distribution of sizes were chosen with the sample distribution being based on the average distribution of packet size on our current campus ethernet backbone. While the filter peaked at about 2.5 Mb/s for the sample distribution, our campus backbone averages around 1.3 Mb/s. Our conclusions from these tests are that the filter is limited by the PC hardware and not the software. The configuration tested was a 33 MHz 486 with two 16 bit SMC ethernet cards on an 8 MHz ISA bus. By simply increasing the speed of the ISA bus to 11 MHz, we saw the performance threshold of the filter on 1514 byte packets jump by 1.0 Mb/s. We feel that 32 bit ethernet cards on an EISA bus would increase the performance greatly.

Memory usage depends greatly on the number of network tables that are loaded into *filter*. At TAMU we currently have one class B IP network that is filtered. *Filter* uses approximately 300 KB of memory in this configuration.

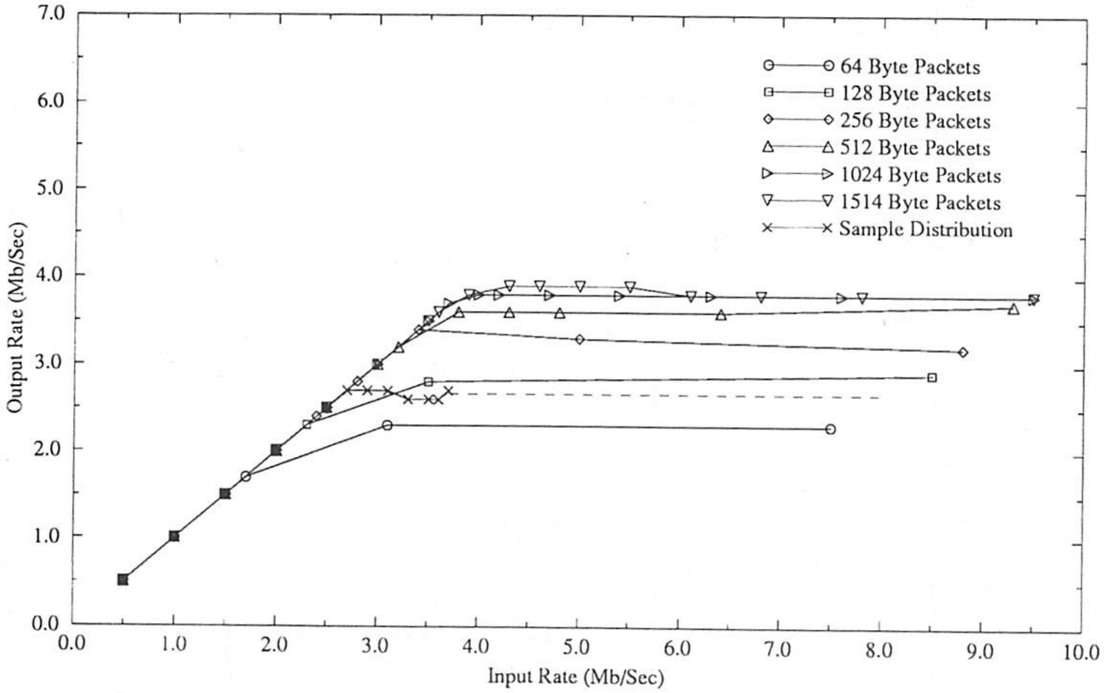


Figure 13. Performance of *filter* under different loads.

5. Monitor

Service Initiation Logging

The first tool, which is actually two tools (*tcplogger* and *udplogger*), records the initiation of a TCP session or UDP session. The start of a TCP session is indicated when the FLAGS field of the packet has the SYN flag set with no other flags set. A record is written to a log file, ASCII or binary, for each session. For a binary log file, the format of the record is:

```
struct sessinit {
      struct timeval start_time;  /* Initiation time */
      unsigned long  ip_src;      /* IP source address */
      unsigned long  ip_dst;      /* IP destination address */
      unsigned long  tcp_seq;     /* TCP sequence number */
      unsigned short srcport;     /* Source port */
      unsigned short dstport;     /* Destination port */
};
```

The TCP sequence number is recorded so that duplicate packets can be removed in a post-processing phase. This simplifies the recording tool.

Detecting the initiation of a UDP session is not as simple. The UDP logging system uses a heuristic to determine the start of a session. Whenever a UDP packet is received, a table of "active" sessions is searched. If the packet belongs to an active session, that session's active time stamp is updated. If no active session is found, then a new "active" session is created and the packet is logged, using the same record structure as used by the TCP logging system. Since UDP packets do not have sequence numbers, the sequence number field is set to zero. After a packet has not been received for a session for an (user specifiable) amount of time, that session is deleted.

Both tools use the SunOS 4.1 Network Interface Tap (nit) with packet filters (nitpf). The use of the packet filter significantly enhances the performance of these two tools.

Service Initiation Log Processing

The binary log files created by *tcplogger* and *udplogger* contain records of all the TCP and UDP sessions that have occurred. This is, of course, a large number. On a normal day, there are over 100,000 TCP sessions and around 50,000 UDP sessions occurring on the TAMU Internet link. A tool was developed for extracting only those records of interest. The tool, *extract*, uses an "awk"-like language for selecting records and printing them. Records can be selected based on source or destination port, host, and network, date, and time. Selectors can be grouped using the boolean "and" and "or" operators. An example *extract* script is shown in figure 14. *Extract* can generate ASCII or binary log files. The binary log file uses the same format as the TCP and UDP logging tools, thus allowing further processing to occur. Since there is no information in the binary log file to indicate whether it is a TCP or UDP entry, the two can not be mixed, and *extract* must be informed of the type of file it is processing.

In practice this restriction is not a problem. There is generally a lot of noise with UDP traffic (traceroutes, FSP, etc.). If mixed together with the TCP log data, a TCP connection might be lost in the noise. Separating them into two log files eliminates this problem. As we normally maintain two logs for this reason, tagging each record by type so that the logs can be combined, is not of significant benefit.

```
#!/usr/local/etc/extract -f
#
#  Print out interesting TCP events coming from the Internet
#
srcnet = 128.194.0.0 {next}  # Skip sessions originating from A&M
dstport = shell ||
  dstport = exec ||
  dstport = 6000 {print; next}
srchost = terminus.lcs.mit.edu ||
  srchost = nyx.cs.du.edu {print; next}
dstport = smtp || dstport = telnet || dstport = finger ||
  dstport = 113 || dstport = ftp || srcport = ftp-data {next}
dstport = nntp && dsthost = news.tamu.edu {next}     | |
dstport = 2000 && (
    dsthost = mud1.tamu.edu ||
    dsthost = mud2.tamu.edu) {next}
dstport > 1023 {next}
{print}  # Print anything that makes it to here
```

Figure 14. Example *extract* script.

Protocol Signature Analysis

While the TCP/UDP logging tools allow us to detect when someone is probing the campus machines for tftp, or some related activity, they don't tell us what happens when someone connects to a system via telnet or some other TCP/IP service. The *etherscan* tool provides this capability. *Etherscan* monitors certain protocols for unusual activities. These protocols are the ones normally allowed through the filtering bridge, i.e., telnet, ftp, smtp. The specifics of what is watched for will not be discussed here, as we do not want potential intruders to know exactly at what we are looking. One example though is attempts to login using system account names, e.g., "root." Another feature of *etherscan* is the ability to detect and report FSP servers. FSP is a UDP based file transfer system which has found favor among those wishing to keep their activities hidden from network and system administrators. A sample listing of the output of *etherscan* is shown in figure 15. At TAMU, there are approximately 20 records recorded per hour. Most of these are due to people attempting to login as the user 'guest' or 'anonymous'.

As with *tcplogger* and *udplogger*, *etherscan* uses the SunOS 4.1 Network Interface Tap and the packet filtering mechanism for performing packet captures.

```
05/26/93 08:35:54.19 [smtp] some.host.edu.2992 HOSTAT.TAMU.EDU cmd help
05/26/93 08:36:00.08 [smtp] some.host.edu.2992 HOSTAT.TAMU.EDU cmd help data
05/26/93 08:36:04.19 [smtp] some.host.edu.2992 HOSTAT.TAMU.EDU cmd help mail
05/26/93 08:36:19.84 [smtp] some.host.edu.2992 HOSTAT.TAMU.EDU cmd help rcpt
05/27/93 11:16:49.44 [udp] AHOST.UDE.EDU.21 possible FSP server.
05/27/93 13:23:30.42 [ftp] 128.194.180.66.27935 out.there.edu attempted login
                           'USER'
05/27/93 13:23:31.87 [ftp] 128.194.180.66.27935 out.there.edu 530 User USER
                           access denied..
05/27/93 13:55:37.29 [telnet] 128.194.225.211.33633 over.there.edu attempted
                              login as 'system'
```

Figure 15. Example output from *etherscan*.

Traffic Analyzer

The final tool, *nstat*, is used to locate changes in network traffic patterns. This tool was originally written in order to gather statistics on the usage of various protocols on the Internet link. By recording these levels on an hourly basis, changes in the usage of a protocol can indicate someone attempting to bypass system security. Using this, an unauthorized FSP server was discovered on the campus when the UDP port that was being used suddenly increased in utilization (this was before support had been added to *etherscan* for detecting FSP servers).

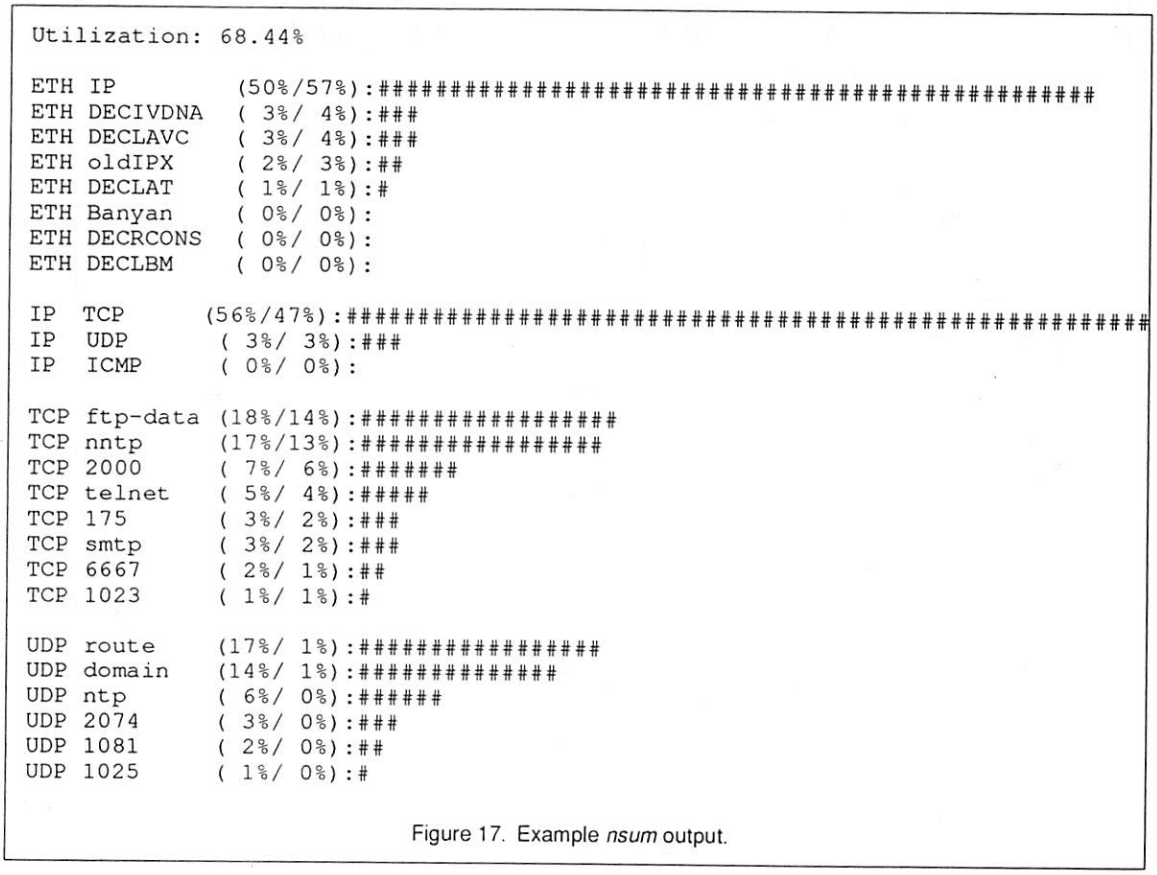
Nstat has a raw output in ASCII format, although it is not really intended for direct viewing. Two programs, *nsum* and *nload*, are provided to analyze the raw *nstat* output. *Nsum* is a PERL program that summarizes the utilizations statistics, giving an ASCII histogram of the top ten usages at the ethernet, IP, TCP, and UPD levels. *Nsum* has support for analyzing certain time periods, such as morning, afternoon, etc., and for selecting weekday versus weekend times. The second analysis program, *nload*, is a simple awk program which produces data suitable for graphing with a tool such as xvgr. A parameter file for xvgr is included. Figure 16 shows a section of the raw *nstat* output. Figure 17 shows a histogram produced by *nsum*.

Ethics

Many may question the ethics and legality of such monitoring. We feel that our current system is not a privacy intrusion. The TCPLOGGER and UDPLOGGER are simply the network equivalent of process accounting, as they log routine network events, but none of the associated user level data associated with the event. Etherscan similarly reports unusual network events, which is the network equivalent of logging failed login attempts.

```
#Start Wed May 19 18:48:01 1993
#Stop  Wed May 19 19:22:52 1993
#550641 packets, 78169154 bytes, 9178 802.3, 2 runt, 3320 missed.
e 600       # 175         b 15882
e 800       # 457927      b 67569221
e 806       # 235         b 14104
e 889       # 184         b 11452
i 1         # 819         b 63686
i 6         # 446119      b 66005034
i 9         # 87          b 104942
i 17        # 10642       b 1375485
t 20        # 40416       b 16027884
t 21        # 1263        b 96426
t 23        # 81314       b 6522485
t 25        # 18540       b 3092486
t 37        # 8           b 480
t 53        # 58          b 4104
t 70        # 11347       b 4266897
t 79        # 262         b 22230
t 113       # 26          b 1596
t 119       # 108098      b 16296687
u 42        # 6           b 360
u 53        # 6634        b 628649
u 111       # 4           b 536
u 123       # 3622        b 325980
u 125       # 67          b 7102
u 127       # 67          b 7102
u 161       # 156         b 13950
u 213       # 24          b 1776
u 513       # 92          b 9960
u 514       # 3           b 348
u 517       # 6           b 552
u 518       # 137         b 17142
u 520       # 4864        b 954544
```

Figure 16. Raw *nstat* output.



```
Utilization: 68.44%

ETH IP          (50%/57%):##################################################
ETH DECIVDNA    ( 3%/ 4%):###
ETH DECLAVC     ( 3%/ 4%):###
ETH oldIPX      ( 2%/ 3%):##
ETH DECLAT      ( 1%/ 1%):#
ETH Banyan      ( 0%/ 0%):
ETH DECRCONS    ( 0%/ 0%):
ETH DECLBM      ( 0%/ 0%):

IP  TCP        (56%/47%):########################################################
IP  UDP         ( 3%/ 3%):###
IP  ICMP        ( 0%/ 0%):

TCP ftp-data    (18%/14%):##################
TCP nntp        (17%/13%):#################
TCP 2000        ( 7%/ 6%):#######
TCP telnet      ( 5%/ 4%):#####
TCP 175         ( 3%/ 2%):###
TCP smtp        ( 3%/ 2%):###
TCP 6667        ( 2%/ 1%):##
TCP 1023        ( 1%/ 1%):#

UDP route       (17%/ 1%):#################
UDP domain      (14%/ 1%):##############
UDP ntp         ( 6%/ 0%):######
UDP 2074        ( 3%/ 0%):###
UDP 1081        ( 2%/ 0%):##
UDP 1025        ( 1%/ 0%):#
```

Figure 17. Example *nsum* output.

6. Machine Cleanup (the tiger scripts)

Structure

The *tiger scripts* consist of one primary "driver" script named *tiger*, a set of scripts which check various components of the system, support scripts, and support files. The driver script, *tiger*, executes each of the component scripts. Its usage is:

```
tiger [-B tigerhomedir] [-d loggingdir] [-w scratchworkdir]
```

The configuration file, *tigerrc*, can be used to enable or disable the different component checks. This allows certain fast executing components to be executed frequently, while other, longer executing components can be executed less frequently.

The driver and component scripts make use of many support scripts and data files. These are used to make the scripts portable. The support scripts are used to set internal variables, such as the pathnames to the UNIX commands used by the scripts, and to convert system configuration files into the formats the main scripts can parse. For example, one of these scripts is used to generate '/etc/passwd'-like input from the various locations that this information is obtained on a particular system. They are also used to provide functionality that a particular system may be lacking. The data files contain information that is compared against information found on the system. For example, one of the files contains the permissions expected on system files and directories.

Component Scripts

The checks performed by the *tiger scripts* are broken out into several component scripts. Ordinarily, these are all executed by the driver script *tiger*, but they can all be executed directly.

Many of the scripts check the ownership and access permissions for files. There are two different types of checks. They will be distinguished in this paper by referring to them in the following manners.

A check of a ***pathname*** means that all components of the pathname are checked. If the support module 'realpath' is available (i.e., it compiled), any symbolic links are handled as well. As an example, on SunOS 4.x systems, '/usr/spool' is a symbolic link to '/var/spool'. Therefore, the pathname '/usr/spool/cron' contains the components '/usr', '/var', '/var/spool', and '/var/spool/cron' which will be checked. All "incorrect" ownerships or access permissions along the pathname are reported in detail.

A check of a ***filename***, ***directory*** or ***file*** means that only the ownership and access permissions of the file referred to by the filename are checked.

The ownership and permissions requirements are user configurable. The default configuration requires that pathnames to system executables and files be owned by root and writable only by root.

check_accounts

The 'check_accounts' script examines user accounts for the following:

- passwordless accounts
- checks home directory ownership and access permissions
- checks ownership and access permissions of configuration files
- disabled account with .rhosts file, cron entries or .forward file that executes a program.

```
--WARN-- [acc004w] Login ID nag is disabled, but has a .rhosts file
--WARN-- [acc006w] Login ID dave's home directory has group
         'apcis' write access.
```

check_aliases

The 'check_aliases' script examines the system mail aliases file. It reports any program aliases, and also checks the pathnames to the executable for proper ownership and access permissions. It also verifies that pathnames to any included files have proper ownership and access permissions (the entries in the included files are also processed).

```
--INFO-- [ali005w] Alias 'drawbridge-server' contains a program
         entry: |"/xpub/secure/mailserver"
```

check_anonftp

This script checks for an anonymous FTP setup, and if one is found checks the integrity of it. Ownership and permissions of critical files and directories are checked. It also reports on any writable directories that are found.

```
--WARN-- [ftp010w] ~ftp/xpub/ftp/bin is writable by 'ftp'.
```

check_cron

The 'check_cron' script checks user 'cron' files. It examines the ownership and permissions of the pathnames used by each cron entry. Executables without absolute pathnames are also reported. The script attempts to be smart and interpret complex cron entries.

```
--FAIL-- [cron003] cron entry for root uses '/var/netlog/bin/logit'
         which contains '/var/netlog' which is group 'wheel' and
         world writable.
         /var/netlog/bin/logit stop
```

```
--WARN-- [cron002] cron entry for root uses '/home/accts/doug/
         tiger-2.1.1/tigercron' which contains '/home/accts/doug/
         shop' which is not owned by root (owned by doug).
         /home/accts/doug/tiger-2.1.1/tigercron -B
         /home/accts/doug/tiger-2.1.1 -l
         /var/spool/tiger -w /var/spool/tiger/work -b
         /var/spool/tiger/bin > /dev/null 2>&1
```

check_exports

This script checks the server side of an NFS machine. Filesystems exported with "root" access, are reported, although the script does understand diskless clients and is (usually) quiet about these. It also reports exporting of the root directory (/), unrestricted exports, etc.

```
--WARN-- [nfs011w] Unprotected directory /fs is exported with root
         access to host(s) HOSTX.TAMU.EDU.

--WARN-- [nfs011w] Unprotected directory /export/export1/sunos5 is
         exported with root access to host(s) thea, and
         poshost.

--INFO-- [nfs010i] Directory /export/export1/root/Xkernel.sun3x is
         exported with root access to host(s) xhost.
```

check_group

The 'check_group' script cross references all of the sources of 'group' information for consistency. Any conflicting group ids or group names are reported. The group files are also checked for correct format.

```
--WARN-- [grp004w] GID conflict for group 'system' between
         /etc/passwd (gid = 2) and NIS (gid = 128).

--WARN-- [grp005w] Groupname conflict for gid 8 between
         /etc/passwd (group staff) and NIS (group grads).
```

check_inetd

The 'check_inetd' script examines the configuration files '/etc/inetd.conf' and '/etc/services.' It looks for things such as mismatched entries (i.e., telnetd on a port other than 23). It also reports any services which have been added from the standard distribution. The pathnames of executables are checked for ownership and access permissions.

```
--WARN-- [inet005w] Service login is using /xpub/etc/in.rlogind
         instead of /usr/etc/in.rlogind.

--FAIL-- [inet009] inetd entry for login service uses '/xpub/etc/
         in.rlogind' which contains '/xpub/etc' which is group
         'daemon' and world writable.

--WARN-- [inet005w] Service shell is using /xpub/etc/in.rshd
         instead of /usr/etc/in.rshd.

--FAIL-- [inet009] inetd entry for shell service uses '/xpub/etc/
         in.rshd' which contains '/xpub/etc' which is group 'dae-
         mon' and world writable.
```

check_known

This script tests for known signs of an intruder. Directories known to be used by intruders are checked for unexpected files. These directories include ‘lost+found' directories, system mail-spool directories and window server directories. The setuid(2) system call is also checked to verify that it is working properly.

```
--ALERT-- [kis001a] /usr/uucp/.sys is a directory."

--ALERT-- [kis002a] /usr/spool/uucppublic/.hushlogin is not zero-
          length."

--WARN-- [kis008w] File ".stuff" in the mail spool, owned by ‘bin'.
```

check_netrc

The ‘check_netrc' script examines the .netrc files in user home directories. It reports if the permissions on the file are incorrect (i.e. world readable). It also reports any entries in the .netrc file which contain passwords which are not anonymous ftp entries.

```
--WARN-- [nrc002w] User imauser's .netrc file contains passwords
          for non-anonymous ftp accounts.

--FAIL-- [nrc001f] User urauser's .netrc file is readable and
          contains passwords for non-anonymous ftp accounts.
```

check_passwd

The ‘check_passwd' script performs the same functionality as the ‘check_group' script, except it works with "passwd" sources. Sources of password information are cross referenced for conflicts and also check for correct format.

```
--WARN-- [pass004w] UID conflict for login ID ‘smith' between
          /etc/passwd (uid = 125) and NIS (uid = 1388).

--WARN-- [pass005w] Username conflict for uid 6 between /etc/passwd
          (login ID sys) and NIS (login ID joeuser).
```

check_path

The ‘check_path' script checks the PATH variable as set in the various shell startup files. It checks for ‘.' (dot) in the PATH, correct ownership and access permissions of pathnames in the PATH and correct ownership and access permissions of executables in the PATH. By default, the script only checks the PATH for the root account. It can be configured to check all users, though the author is not comfortable with this.

```
--INFO-- [path008i] Setuid program /usr/bin/uux in root's PATH from
          .cshrc is not owned by root (owned by uucp).

--WARN-- [path002w] /usr/bin/ls in root's PATH from .profile is not
          owned by root (owned by bin).

--INFO-- [path006] The PATH set in root's .profile contains
          ‘/usr/bin' which contains ‘/usr' which is not owned by
          root (owned by bin).
```

check_printcap

The script ‘check_printcap' checks the pathnames to filters used by printers for proper ownership and access permissions. Since not all systems use the BSD print system, this script is invoked only for systems on which it is used (there currently isn't an equivalent System V print system checking script).

```
--WARN-- [pcap001w] Print control ‘if' for printer ‘lw2' uses
```

```
        '/usr/local/lib/topsif' which contains '/usr/local/lib'
        which is not owned by root (owned by bin).
```

check_perms

The 'check_perms' script checks the ownership and permissions of system files. A database (specific to the platform) describes which files to check and what the expected permissions are. Files and directories such as '/', '/etc', '/etc/passwd', '/etc/group' and '/etc/aliases' are a few examples of the many files which are checked.

```
--WARN-- [perm019w] The owner of /etc should be root.
--WARN-- [perm019w] /etc should not have group write.
--WARN-- [perm003w] /sbin should not have group write.
--WARN-- [perm003w] /usr should not have group write.
```

check_rhosts

The 'check_rhosts' script examines the .rhosts files in user home directories. It checks the permissions on the files and examines the files, reporting entries with a '+', attempted comment entries, entries that are only partially specified. This is a remote hostname with no remote username. There is no direct security problem here, but it can lead to one as people carry a .rhosts file from site to site.

In addition, it is possible to configure the script to report hostnames that do not match a set of regular expressions. A version of this script written in PERL is provided as well which attempts to verify that the remote user is the same person as the local user (using 'finger').

For large sites, with user home directories distributed across multiple NFS servers, it is possible to configure this script to only check accounts with home directories on local filesytems. This can greatly increase performance. This can also be done for 'check_netrc' and 'check_accounts'.

The '/etc/hosts.equiv' file is also examined. All trusted hosts are reported. Any included netgroups are expanded and reported as well.

```
# Checking accounts from /etc/passwd...
--WARN-- [rcmd004w] User snag's .rhosts file has a '+' for user
         (host ourhost.tamu.edu).
# Checking accounts from NIS...
--WARN-- [rcmd006w] User imauser's .rhosts file has group 'tamug'
         and world read access.
--WARN-- [rcmd006w] User urauser's .rhosts file has group 'onet'
         read access.
```

check_signatures

This script is used to validate system binaries. It does this through the use of a data file which contains digital signatures, generated from distribution media, for important system binaries. There is currently support for two different signature methods: the XEROX Secure Hash Function signatures, commonly referred to as Snefru, and the RSA Data Security Inc., MD5 Message Digest Algorithm. An output block size of 8 is used (256 bits) for the SNEFRU hash. The script auto-detects the type of signature and generates the appropriate one for comparison.

The script reports any system binaries which do not match the stored signatures. It also reports system binaries which are out of date in regards to security patches (using signatures generated from the replacement binaries in the patches). This provides a means of determining quickly whether critical security patches have been installed on a system. A system is being planned which will allow up to date signature databases to be retrieved from a central site(s).

```
--WARN-- [sig004w] None of the following versions of /usr/bin/login
         (-rwsr-xr-x) matched the /usr/bin/login on this machine.
         >>>>>> Sun Patch ID 100630-01
         >>>>>> Sun Patch ID 100631-01
         >>>>>> Sun Patch ID 100633-01
         >>>>>> SunOS 4.1.2 (security patch is 100630)
--WARN-- [sig015w] /usr/bin/mail is from Sun Patch ID 100224-03
         (current is 100224-06)
--WARN-- [sig004w] None of the following versions of /usr/etc/
         rpc.yppasswdd (-rwxr-xr-x)matched the
         /usr/etc/rpc.yppasswdd on this machine.
         >>>>>> SunOS 4.1.2
```

find_files

The 'find_files' script searches through the file systems and locates files that might present a security problem. These are

- setuid executables
- device files
- symbolic links to system files
- world writable directories
- files with an undefined owner or group
- files with unusual filenames

Setuid files are checked to see if they are scripts, or if they contain relative pathnames (an admittedly crude check). It also compares the list of setuid files against a list prepared from distribution media and reports any new setuid programs.

Any device files found in non-standard locations (i.e., not in /dev) are reported. The script understands diskless clients and will automatically ignore the device directories for these as well. Other directories can also be added to this list by setting a variable in the configuration file.

Any symbolic links to system files (such as /etc/passwd) are reported as well. While not directly a security problem, an ill-placed 'chown -R' or 'chmod -R' (or equivalent) could create one.

World writable directories are reported, as they are often used by intruders as a place to store log files. Unowned files are often an indication of a break-in. They also can lead to unexpected access for an account.

The unusual file names includes files with spaces in them, leading '.' or other characters. The list of file names is customizable via a variable which can be set in the configuration file.

check_embedded

The 'check_embedded' script examines files and extracts any apparent pathnames which are embedded in the files. These pathnames are checked for "proper" ownership and access permissions. These files indicated by these pathnames are then in turn checked for embedded pathnames. This process is continued until no new pathnames are found, or a user specified search depth is reached.

The initial list of files searched comes from two sources. The first is a static list specific to a platform. For example, on SunOS 4.x systems, this would include the /etc/rc.* scripts. The second source is from the other 'tiger' scripts. When the scripts are run from 'tiger' or 'tigercron', they generate lists of files that should be checked. For example, 'check_path' will request that any executable in root's path be checked, and 'check_inetd' requests that all the servers defined in /etc/inetd.conf be checked.

```
--WARN-- [embed002w] Path '/usr/sbin/fsck' is not owned by root
         (owned by  bin).
         Embedded references in:
         /sbin/mountall->/etc/init.d/MOUNTFSYS
         /sbin/mountall->/etc/init.d/buildmnttab
         /sbin/mountall->/etc/init.d/nfs.client
--WARN-- [embed003w] Path '/usr/sbin/ypinit' contains '/usr/sbin'
         which is group 'bin' writable.
         Embedded references in:
         /usr/lib/netsvc/yp/ypbind->/usr/sbin/sysidnet->
         /etc/init.d/sysid.net
```

Miscellaneous checks

In addition to these standard checks, miscellaneous checks specific to a system are also performed. Items such as the use of the 'securenets' file on SunOS NIS servers and the removal of the "_writers" property for printers on NeXTOS are examples of the checks performed.

```
--WARN-- [misc004w] The PROM monitor is not in secure mode.
--FAIL-- [misc003f] No /var/yp/securenets file.
```

crack_run

The script 'crack_run' is used to perform password cracking. No password cracker is provided with the 'tiger' system. It is expected that a tool such as Alec Muffett's 'Crack' will be used. 'crack_run' collects all password sources and runs the password cracking tool on them, and reports the results. Since this can take days to complete (or longer), 'tiger' by default does not wait for this to complete.

```
--WARN-- [crk001w] The following login id's have weak passwords:
         imauser urauser
```

Isn't this just COPS?

One common question is how does 'tiger' compare to the COPS package by Dan Farmer. There is a lot of overlap between the two packages. Much of this is intentional. Dan Farmer allowed us to borrow ideas and code from his COPS package. We are giving Dan (and anyone else) the same ability in regards to 'tiger'.

There are advantages and disadvantages between the two packages. Note that most of this is subjective and hence will no doubt be biased.

We feel that 'tiger' is easier to use than COPS for a person who is not familiar with systems administration. As stated earlier, once unpacked, all that is necessary is to run 'tiger' and a report will be generated. The explain facility allows the administrator to get a better idea about the entries in the report. One area that 'tiger' is lacking in is user documentation.

Another area we feel 'tiger' is at an advantage is based on part of the design structure, in which no system files are referenced directly. This feature is what allows the auto-checking of multiple information sources (for example /etc/passwd, NIS, NetInfo, etc) without having to alter the scripts which perform the checks. This makes running the scripts easier for the administrator.

There are other areas that we feel 'tiger' is better than COPS. These include the use of digital signatures for checking for security patches, the complete checking of pathnames containing symbolic links, and the more thorough examination of system configuration files.

We recognize that COPS does have advantages over 'tiger'. The primary one is that COPS is a "proven product". It has been used by thousands for two to three years now. Also, though 'tiger' will attempt to run on platforms for which no configuration files exist, COPS is more likely to succeed in running. The use of newer shell features, primarily shell functions, prevents the use of 'tiger' on any UNIX system which has an older Bourne shell. There are also miscellaneous checks performed by COPS that have not been integrated into 'tiger'. COPS also includes the 'kuang' expert system checker. There is no equivalent functionality in 'tiger'.

7. Observations

We have been using this combination of filtering, monitoring, and checking almost a year, with very positive results. While our network monitoring tools continue to show incoming intrusion attempts (unfortunately along with some outgoing attempts), we have had no major incidents of the type we experienced last summer. The combination of approaches seems to have struck an appropriate balance between security and availability for our academic environment.

Our monitoring tools have produced some interesting statistics. During the last four weeks, for example, we have observed the following number of incoming security events:

e-mail forgery	6
knob-turning	48
TCP attacks/events (X11, DNS zones, rshell ...)	30
UDP attacks/events (TFTP, SNMP ...)	7

Of these 91 incidents, 49 (or 54%) originated from .edu sites. Edu sites account, however, for only 30% of all internet hosts (according to the July 1993 Internet Domain Survey). This means that a minority of hosts are accounting for a disproportionate majority of intrusion activity. One would hope that other university networking groups would be actively trying to reduce these incidents, yet of the requests for the etherscan tool, less than 25% have come from university sites.

8. Conclusions

A set of policies and tools for filtering, monitoring and checking has been developed in response to a significant series of intrusions from internet. Each of these three areas has proved critical: the filtering for its ability to protect machines from attack, the monitoring because it augments the filter and has yielded significant information about the intruders and their methods, and the checking tools for their ability to automate the task of checking and cleaning a large number of machines. With these tools and associated policies, we have achieved an appropriate balance between security and availability in an academic environment.

9. Availability

Drawbridge, the *tiger scripts*, and all monitoring tools other than *etherscan* are now available via anonymous ftp in sc.tamu.edu:pub/security/TAMU. Due to export restrictions, the DES routines used in *drawbridge* have been put in a separate tar file and are readable only by U.S.A. sites. Other sites should have no problem either running the filter without encryption or dropping in their own favorite encryption package.

The distribution of *etherscan* has been hotly debated within the TAMUSC group. One argument is that *etherscan* should be freely released, as the crackers already have equivalent knowledge and tools (they do) and restrictions would only hurt valid administrators. The counter argument is that free availability of the intrusion signatures would enable the crackers to design better intrusions and the availability of sources would provide novice crackers a significant help. Our resultant compromise will be to provide copies to Network Information Center registered site contacts, given an official request on respective letterhead. Requests should be sent to:

Dr. Dave Safford, Director
Supercomputer Center
Texas A&M University
MS 3363
College Station, TX 77843-3363

10.References

[1] D.B. Chapman. Network (In)Security through IP Packet Filtering, *Proceedings of the Third UNIX Security Symposium*, September 1992.
(available from ftp.greatcircle.com as pub/pkt_filtering.ps.Z)

[2] Ranum. "Thinking about Firewalls", available on the Internet

[3] Violino, Bob. "Are Your Networks Secure?" Information Week, April 12, 1993, page 30.

UNIX® Security Update

Jerry M. Carlin
Pacific Bell
2600 Camino Ramon
Room 3N750JJ
San Ramon, CA 94583
jmcarli@srv.pacbell.com

ABSTRACT

This presentation reviews key elements of the past few years work at Pacific Bell enhancing the security of UNIX®[1] systems. The paper includes a review of the central elements of our strategy and a discussion of changes that have taken place over the past couple of years.

The critical parts of this effort have been developing security standards, arranging for appropriate training, selecting and developing security and auditing tools, working on influencing vendors to implement C2 security, writing security sections of volume purchase agreement RFPs, and consulting with application developers and system administrators.

Key changes to our approach in the recent past includes integration of tools into the UNIX standards, evolving methods of securing of our Internet gateway and the use of tools in investigating suspected intrusion attempts. Also key is managing remote network and dial access and starting to develop approaches to using emerging technology such as the OSF DCE and POSIX P1003.6.

1. Introduction

Since the formation of the Pacific Bell Interdepartmental Data Protection organization and the start of systematic work securing UNIX in 1988, significant changes have occurred in the UNIX environment. Vendors have been seriously addressing UNIX security. Many security tools have become available and Pacific Bell has been involved in significant work creating standards, implementing system security and developing a clearer picture of application security requirements.

Tools and techniques are valueless without proper motivation. We have mounted a major effort to inform employees that security is not only important, but that it is an integral

[1] UNIX is a registered trademark of USL International

part of the job. Videos have been produced that detail some problems we have had and booklets have given hints on such matters as handling strangers without IDs and developing good passwords.

2. UNIX Security Progress

This section gives an overview of the most significant changes to UNIX security at Pacific Bell including the versions of UNIX available, training, tools, dial access and other measures. Also, a review of the measures in-place on our Internet gateway machine offers a full review of security measures we are using.

When the effort first started in 1988, the security of UNIX systems was haphazard at best. No standards existed; the versions of UNIX then available had many bugs and design flaws; people assumed that no one could find our phone numbers so dialup security was non-existent; no security tools existed; groups of people typically shared user IDs; some IDs did not even have passwords and in some systems it was typical that all the users had the root password. The situation has improved since then!

2.1 UNIX Versions

Vendors have significantly enhanced UNIX. The appearance of "C2" UNIX has given us features including a security manual, shadowed passwords and kernel-level auditing. Some B-level enhancements have also become available in some versions and these include such useful features as access control lists. In addition, the POSIX security section (1003.6) has been through two ballot rounds and is hopefully reasonably close to finalization. Finally, the federal government through the NIST's "Minimum Security Functionality for Multi-User Operating Systems" work as well as the new Federal Criteria draft are starting to drive vendors to implement useful security upgrades.

In addition, security bugs have received serious attention from a number of vendors. For example, SUN created a 'Customer Warning System' and has released a number of critical security patches. This effort has been facilitated by the Computer Emergency Response Team's role in coordinating handling of security problems.

We have also incorporated security in the RFP process. Two recent RFP's included questions on plans for using forthcoming standards such as POSIX 1003.6, implementation of OSF's DCE and methods for handling security problems.

2.2 Tools

Bellcore has produced a very full-featured and integrated set of UNIX security tools for the use of the regional Bell operating companies. This toolkit includes auditing for changes to system files, checking for configuration problems, replacements for system programs such as login and a security manager. This toolkit consists of locally developed programs and integrates public-domain software such as COPS and log_tcp.

With this Toolkit it is now possible to do a thorough audit of the security state of a machine as well as to reliably discover any unauthorized changes to critical files and directories.

2.3 Training

In 1988, available learning resources was limited to one book, *UNIX System Security*, by Wood and Kochan whereas now several texts are available.

Security training has been made available via a course offered by Bellcore and we have also suitcased an AT&T UNIX security course.

2.4 Remote Access

One of the most critical areas we have been working in has been securing remote access to our systems. Use of SecurID®[2] technology for dial access is a key element of this work. We have installed these systems on our TCP/IP network, our UB-LAN terminal network, on DataKit®[3] and on a number of machines individually.

2.5 Application Security

We have spent a significant amount of time and effort to develop and to integrate security as part of the application planning and development process. Security questions are part of our application review process. Three rounds of UNIX security reviews have been done to determine the state of security and to assist in the planning process for security enhancements. We have also participated in the requirements and design phases of a number of applications in a consulting role.

2.6 Individual User-IDs

Along with other operating environments such as MVS, large systems used by tens of thousands of employees such as the infamous *COSMOS* were converted to run with individual IDs rather than group IDs, or, in some cases, with no user IDs or passwords. This involved modifying the personnel data system to create a unique, consistent ID for all employees, implementing of mechanisms to handle contractors and vendors, changing applications to handle individual user IDs and creating of procedures for granting access to applications. In one case the software also had to take into account moving of users between machines based on capacity considerations.

2.7 Security Administration

A start has been made on coordinating and rationalizing security administration by making available to UNIX administrators a list of current employees extracted from the personnel system and current contractors as maintained by an user administration system.

2.8 Internet Firewall

A key element in our strategy is the use of a "firewall" to manage Internet access. This has provided us with a system that is a model for implementing reasonable security

[2] SecurID is a registered trademark of Security Dynamics Technologies, Inc.

[3] DataKit is a registered trademark of AT&T

measures. It has served as a test-bed to determine which standards are meaningful and has helped shape the boundary between the needs of various applications and meaningful security measures. Key measures adopted include:

- Basic security measures:

  Of course, we implement basic UNIX security measures. For example, /usr is typically mounted read-only. Setuid programs are forbidden on the filesystem containing users and on the root filesystem. Password aging is enforced and C2 shadowed passwords enabled.

- Bellcore Security Toolkit:

  The latest release of the Bellcore toolkit is implemented on the machine early on in the implementation process. It helps provide feedback to Bellcore and guidance in selecting which options we want to use locally.

- log_tcp:

  This program allows selective rejection of service requests from specific sites as well as auditing tools such as a reverse finger on telnet/rlogin requests. Currently we allow but reverse finger incoming telnet requests and refuse any requests from "unknown" locations. We also deny all other service requests from the Internet including the r* commands. The /etc/hosts.allow command we use to log telnet requests is:

  ```
  in.telnetd:    ALL: (/usr/ucb/logger -p daemon.notice "*** %d from %h ***"; \
                   /bin/date; /usr/ucb/finger @%h; echo "") >>/var/log/%d &
  ```

- SUN SHIELD™ (ASET):

  SUN has a security add-on package, *SHIELD*, which helps track changes to system resources, report on configuration options and apply additional controls to access including time of day and port access. Due to compatibility issues, we are using only the reporting features. Also, since some of the capabilities duplicate that available with the Bellcore toolkit, only those items that are unique to SUN are tracked.

  We have modified the shell scripts to keep a pointer to the current and previous reports and only show differences between them except for once a month when the entire set of reports is reviewed.

- ftp tracking:

  We have installed a better ftp, *wuarchive ftp*, which allows differentiation of local versus remote ftp users. This allows us to give ftp access to our vendors and other partners while at the same time allowing for a more secure access for employees.

- security patches:

  We try to install security patches in a timely manner. Monitoring various lists enables us to find out about the patches as soon as they become available. We have also scanned SUN's *SunSolve* CD-ROM archive to verify we have all critical patches. These patches include specific kernel and utility fixes as well as a patch to correct problems in system file permissions.

A database of patches is maintained so other administrators can get them locally via ftp.

- C2 auditing:

  Two key things we audit for are invalid login attempts and major administration events. A significant number of invalid login attempts are found and this enables us to offer feedback to offending sites. Due to our network connectivity, we notice a fair number of problems which could be characterized as "doorknob rattling". These are attacks that look as if someone is wandering "down the street" looking for an "open door". Here is one typical sequence:

  ```
  July 10 15:32:16 1993,login help,abcplan.abcd.edu
  July 10 15:32:33 1993,login guest,abcplan.abcd.edu
  July 10 15:32:47 1993,login new,abcplan.abcd.edu
  July 10 15:33:00 1993,login help,abcplan.abcd.edu
  ```

- Internet router:

  The router connecting us with the Internet has been configured to block the worst problems. The services and ports blocked are SUN-RPC (UDP/TCP 111), NFS (UDP 2049), LPD (TCP 515) and TFTP (UDP 69).

- White Pages:

  One example of the balance between security and usage is our system for making selected information about employees available to the Internet. Based on the existing ability to lookup employee telephone numbers via touch-tone access, we allow access to employees name, email ID and phone number from the Internet. This is done by replacing the standard finger and whois programs with a special purpose utility. Since the full database includes proprietary information, this gave us a way of satisfying business needs without allowing unrestricted access to sensitive information.

  This utility checks to make sure the query is limited by scanning for characters such as '=' and '*'. This forces queries to be only on name or email ID. The front-end utility then calls another utility, *ph*, and returns only name, email ID and phone number. In addition, each use of this utility is logged.

  Here is an exerpt from the code:

  ```
  $_ = <>;
  chop;
  s/.$//;
  $whois=$_;
  s/.*=//;
  s/\*//;
  ```

```
if ($whois eq "") {
        system "logger -t in.whois -p local7.info ' $cpuname $from : ";
        printf("only individual lookups supported0);
        exit;
}

if ($whois ne $_) {
        system "logger -t in.whois -p local7.warning ' WARNING $cpuname
                $from $port : $whois $_'";
        printf ("error in request: request rejected0);
        exit;
}
```

- Proxy Programs:

  Another example of this balance between security and functionality is our use of a SUN consulting special, *Igateway*. *Igateway* allows a user to "tunnel" trough a firewall in specific directions and thus access to the Internet via ftp and telnet. Therefore it gives us the ability of internal users to gain external access without allowing external users internal access.

  We also installed the *tcpr* package which provides the same services but which is perl-based and thus allows implementation in more environments.

- syslog:

  Very extensive use is made of the *syslog* capability. Logging is done to a remote, centralized host for greater security and convenience. A public-domain perl program, *swatch*, is used to scan the log file to extract items for later perusal. Rather than pick only some events we want to monitor, we chose to exclude events we did not care about so we increase the odds of seeing a relevant event. Typically this report is two to three screens long and we take a few minutes to review it daily.

  This log also helps us detect normal system administration problems. Here are a few examples from a recent log:

  The first shows a series from a single site. I believe the incident was caused by a company employee who had forgotten his password:-)

```
Aug  5 20:58:08 gw.PacBell.COM in.rlogind[12551]: refused connect from
asd.Berkeley.EDU

Aug  5 21:00:41 gw.PacBell.COM syslog: in.telnetd from asd.Berkeley.EDU

Aug  5 21:11:30 gw.PacBell.COM syslog: in.telnetd from asd.Berkeley.EDU

Aug  5 21:23:52 gw.PacBell.COM syslog: in.telnetd from asd.Berkeley.EDU

Aug  5 22:47:58 gw.PacBell.COM in.rlogind[21813]: refused connect from
asd.Berkeley.EDU
```

```
Aug 5 22:48:10 gw.PacBell.COM syslog: in.telnetd from asd.Berkeley.EDU

Aug 5 22:48:37 gw.PacBell.COM syslog: in.telnetd from asd.Berkeley.EDU
```

We also log and refuse ftp attempts a site the nameserver cannot resolve:

```
Jul 31 13:48:17 gw.PacBell.COM ftpd[25915]: FTP LOGIN REFUSED
(access denied) FROM 35.195.34.6 [35.195.34.6], ftp
```

We see normal administration messages:

```
Aug 1 00:59:17 gw.PacBell.COM nntplink[222]: gazette.tandem.com: Link
down for 12 hours

Aug 2 02:00:08 gw.PacBell.COM syslog: ACCT ERRORS : see
/var/adm/acct/nite/active

Aug 2 16:10:20 gw.PacBell.COM sendmail[15994]: Unparseable user
<ba.bad@@PacBell.COM> wants to be <ba.bad@@PacBell.COM>
```

Here is a "normal" attept to connect to a service we disallow from outside:

```
Aug 1 16:40:37 gw.PacBell.COM gopherd[4716]: refused connect from
peach.abc.edu
```

Then there are the ones that we look at more carefully. The first we would have already seen in the C2 log and the second is a warning from our *whois* program that a disallowed wildcard was attempted. We disallow wildcards since otherwise someone could use the database to compile a list of all Pacific Bell employees and this is something we do not allow. We do see a fair number of lookups on "Smith", however.

```
Aug 1 22:47:11 gw.PacBell.COM login: REPEATED LOGIN FAILURES
ON ttyp1 FROM puke.abcd.cszztt, _

Aug 3 10:15:44 ns.PacBell.COM in.whois: WARNING abcd.ZQR.COM
163.145.34.154 2371 : fos* fos
```

3. Current Problems

Even though we have made a good start, there are several key areas needing continued attention. The biggest consideration is no surprise: prioritizing security requirements given budgetary constraints.

3.1 Remote Access

We have not yet eliminated all unsecured regular modem access. Also, not only is modem access still an issue, but the proliferation of newer networking technologies such as ISDN, SMDS and other public packet switching networks have to be addressed. In some cases using the "closed user group" feature of such networks to restrict access has been helpful.

3.2 Obsolete hardware/software

We still have a lot of obsolete hardware and operating systems. Replacing this infrastructure is underway but economic considerations makes this a slower process than would be ideal from the security standpoint.

3.3 Client-server programs

Our use of distributed application technology is growing quickly. We therefore need Kerberos V5 and/or OSF DCE now so we can stop scripting passwords to applications and have better end-to-end security in general. Use of OSF DCE offers the possibility of significantly enhancing application development by replacing the need for terminal-emulation applications with a data-centered interface.

3.4 Security Administration

With downsizing and restructuring, training and motivation of security and system administrators needs to be continually addressed. We are working to make security part of the formal performance review of people who have security responsibilities.

3.5 Turnkey Applications

Currently we are working with vendors of major applications to ensure that we have sufficient security. We need to overcome the process of having to negotiate such measures as loading the Bellcore toolkit on an application-by-application basis and move to having an understanding of our security needs on the part of vendors of major systems.

3.6 Common Attacks

We have to investigate and respond to a continuing series of potentially security relevant attacks. Fortunately most of these are either "doorknob rattling attacks" or incidents that look very much like a naive user that does not understand the tools. These include someone trying a *whois* command that returns "no one found" followed by a login attempt to *whois*. Because we have been a backup server and listed in the NIC WHOIS database, people make incorrect assumptions about what being a backup Internet WHOIS server implies and we see requests for such things as Bill Clinton's email address and lists of users on a university computer.

We also see incidents that look like an attempt to see if we have services such as *gopher* and *wais* available. These are characterized by a login attempt to one or more of these common IDs.

It is interesting to note that these events come from the .COM domain as well as .EDU, although, as one might expect, more come from educational institutions.

4. Current Activities

Shortly a new version of our security standards will be published accompanied by a new security review questionnaire. We are making a major effort to distribute the newer version of the Bellcore toolkit because of the significant enhancements over the prior versions. In addition, we are closely tracking the industry and Bellcore developments with distributed computing, Kerberos and OSF DCE and hope to start formulating implementation plans shortly. Also, a major effort is underway to enhance the security in the public switched network and this includes the security of UNIX processors which are a part of that environment.

4.1 Internet Access

For obvious reasons we plan to implement use of SecurID tokens for telnet from the Internet. Also, as use of the Internet grows, we face a growing challenge to make new services available without compromising essential security. We are evaluating the use of screening router to allow one-way access to services such as WAIS, WWW, Gopher and Archie. This would allow us to make these services available faster since we would not need a proxy server for each new service.

4.2 Standards Update

We have learned that written standards should help people clarify priorities and activities as well as give specific guidance. Therefore, the following exposition of key changes to our standards will help clarify our current plans and priorities.

- Fundamental Concepts:

  This section was added to help people realize the scope and purpose of security efforts. It is intended to indicate that security is a large topic and is not limited to choosing a good password.

  > "This document instantiates a number of fundamental security concepts. These include assurance of adequate authorization, authentication, audit and administration of a system. The standards are intended to implement a reasonable "least privilege" policy whereby users get all the resources needed to perform their duties but not full control over the system. For example, the number of people with "superuser" privileges must be limited to those that need it as part of their duties."

- In A Nutshell:

  This topic gives basic guidance about priorities:

  > "Firstly, protect any public network (dial, Internet, PPSN) access using approved mechanisms, typically using SecurID® cards for interactive users and gateway machines for host-host traffic. Secondly, authorize all users and enforce password standards from the Baseline. Thirdly, implement a reasonable "least privilege" policy to ensure that what users can do is limited by what they need to do their job. Fourthly, audit for compliance and to detect problems and unauthorized changes."

- Starting Point:

  Given the progress of the industry, we tried to clarify that the starting point for security was implementing what the vendor provides. We have had questions about whether to use our guidance or guidance from a specific vendor in implementing security so this section clarifies our view:

> "Use what your vendor delivers as the starting point for implementing security. For example, file and directory ownerships and permissions should not be looser than what the vendor delivers but might need to be tighter to implement a reasonable "least privilege" policy. Vendor manuals must be consulted to determine specific security recommendations above and beyond those given here."

- Key contact list:

  In a very-large company such as Pacific Bell, where applications and machines are distributed all over the state, it is very important to know who to contact if something goes wrong so the document states:

  > "Administrators must communicate emergency contact information to IDP including name, phone, pager, email address, fax number and scope of responsibility including system or applications. They need to monitor security related communications including the IDP security mailing lists and the netnews groups such as pb.security and comp.security.announce for relevant information."

- Tools:

  We have found that it is much easier to have a tools orientation to security. Suggesting that certain tools be used and explaining the reasoning behind it helps a number of users. The introduction to this concept says:

  > "Tools that are acceptable to accomplish the tasks specified by this document include those delivered with UNIX including C2 facilities, the Bellcore UNIX Toolkit, others available through the UNIX/C library, any specified in the approved product list, selected public-domain programs as well as locally-developed programs. Contact Interdepartmental Data Protection for information on site-licenses."

- Auditing:

  A significant change in the standards document was the reorganization of the auditing section. Along with a statement of purpose was a suggested auditing schedule:

  > "The purpose of this security auditing is to find any problems with the configuration of the system such as bad permissions, to monitor for unauthorized changes to the system or application and to detect intrusion attempts. Note that this list must be supplemented with any needed DBMS and application-level auditing. Any auditing plan other than the one suggested here must be documented including the rationale for what is being audited and why."

  > "The tasks are divided into installation or initial tasks, daily, weekly, monthly, quarterly and annual tasks... The software package or

capabilities given are as follows: "C2" indicates a capability available with most C2 implementations, "syslog" is for systems supporting the syslog function call, "bcr" refers to Bellcore toolkit software, "local" is for locally developed tools available from the UNIX/C library and other packages are listed by name such as "swatch" and "log_tcp". Other packages that can be used include "cops", SUN's ASET, for which we have a site-license, and third-party products."

- Other:

  Some other sections we dealt with was the specifying of "industry standard" encryption, suggestions that OSF DCE or Kerberos should be used where appropriate, firewall machines should be used to connect to public networks and such tools as TCP wrappers like *log_tcp* should be used.

4.3 Security Questionnaire

As has been mentioned, a security questionnaire has been an integral part of the process of improving UNIX security. Here are some key questions from the latest version that reflects the revised standards.

- Give name of the application, acronym and business purpose.
- Is all dial access protected via SecurID?
- What security training have you received?
- Has "C2" been activated? Which features?
- How many people have the root password?
- Which security tools are being used?
- How has the 'no trespassing notice' been implemented?
- Are there any group IDs?
- How are vendor IDs controlled?
- Is password aging implemented?
- What were the last security patches installed, when were they installed, how did you hear about them and where did you get them from?
- Do you have DES encryption installed?
- What security auditing is being done?
- Are there any application-based audit trails?
- Are file and directory permissions the same or tighter than delivered by the OS vendor?
- Describe the purpose of any setuid root programs besides those that came with UNIX?

- What networks does the system/application use?
- What network security measures are in place?
- Describe any connectivity with non-Pacbell systems?
- Describe any significant changes/upgrades to security in the past year.

5. Conclusion

In the past several years we have seen significant enhancements in UNIX security, holes have been plugged and new functionality has become available. We now have a good set of tools and training available to help administrators with their job. Standards and questionnaires have been refined to assist administrators and developers. I believe it is now possible to have the same level of security on UNIX as on any other commercial operating system.

But, the biggest remaining problem is the same now as in the past: making sure security is viewed as a critical business success factor with the result that security is viewed as a required part of the job of every system administrator and application developer.

The Persistent Hacker: An Intruder Attacks A New Internet Host

Eduardo Rodríguez José M. Piquer
erodrigu@dcc.uchile.cl jpiquer@dcc.uchile.cl

Departamento Ciencias de la Computación,
Universidad de Chile. Blanco Encalada 2120
Santiago, Chile.

August 14, 1993

Abstract

As newcomers to the Internet, network security was considered a minor problem in the whole set of services and programs that we enjoyed setting up: the primary name server for Chile, the main news feed (almost full) for the country, the local workstation network design, to mention just a few.

Moreover, even if we heard a lot of stories about the activities of hackers in the Internet, they all seemed far away from our country, surrounded by mountains and the sea at the end of the world. And our computers are mainly used by students and professors in a supposedly secure academic environment. Our confidence was completely misplaced.

In this paper, we describe the activities of a hacker in our hosts during the last few months of 1992 and the beginning of 1993, and the conclusions and experiences he (she, them?) left us with. This was our first serious hacker problem, with an intruder that had only two powerful weapons: time and patience.

We named that hacker "Morgan", because the Chilean coasts were devastated more than two centuries ago by an English pirate of that name when our country was a Spanish colony (nothing personal against England :-) He entered the campus network from many sites (always hacked sites) and from here to other hosts in many other countries.

During the time he was our uninvited guest, he showed perseverance and regularity in his procedure. Based on that, we can affirm that he used the same method to attack other hosts. We obtained evidence of this from only one other host, located in Europe. In general, system administrators don't like to talk about their security problems.

This paper has two authors, however it is a report of the work of the whole system administration staff of the Department of Computer Science.

1 Introduction

This paper is intended to present a detailed trace of a hacker's behavior. We were alerted by European friends that our machines were probably being used by an intruder in December 1992. To our surprise, we found out that he had entered all of our machines, and that he was our guest for three months with root access! At this moment, we realized that our lack of concern for security was a big mistake, and we began to monitor the hacker, to learn his ways and to find out how he managed to enter our machines.

This paper is divided as follows: Section 1 presents a brief history of our department. Section 2 shows how Morgan enters a machine and how he gets root access. Section 3 describes what he does after obtaining root access. Section 4 tries to figure out Morgan's objectives. Section 5 details our procedures and traps set up to track our hacker. Finally, section 6 presents some conclusions.

2 History

In 1985, the Department of Computer Science of the University of Chile established an experimental UUCP network, using slow modems and phone lines, linking together three Unix systems from different Universities. Very quickly a connection to the international UUCP network became an important goal. A test period (during 1986) with an X.25 connection to `inria` in France, and to `seismo` in the USA ended when we started using Telebit Trailblazers to call long distance to `uunet`. At that time, we registered the Chilean top level domain (`.CL`), and decided since the beginning to use a domain name system. This Chilean UUCP network slowly grew as more Universities, companies and private individuals joined it (with currently more than 50 nodes), but its growth was stunted by the high cost of international telephone communications. In 1992 Chile joined the Internet through two dedicated 56Kbps satellite links. Our department was deeply involved in the technical issues surrounding this connection, setting up most of the initial TCP/IP services in Chile: mail, news, and the primary nameserver. Security was our last concern.

As of may 1993, the whole (Internet and UUCP) top-level domain includes 844 nodes and 32 direct sub-domains.

3 Arriving to Port

The first thing Morgan did, was to obtain the password of one of the users of our host, presumibly the password of one of our professors doing postgraduate studies in the USA or Europe (at that time, the Department of Computer Science had more than half a dozen professors studying abroad). He entered the system at hours of low activity or when nobody was around (primarily at night or in the weekends).

He placed trojan horse commands throughout the system that recorded the password of the root superuser. The most common trojan horse was the `su` command implemented as a Bourne shell that records the username and password in a hidden file, and executes the real `/usr/bin/su` after that. He placed the `su` trojan horse in user's directories, usually the $HOME/bin, that in most cases is in the $PATH variable. It is very difficult to detect the trojan horse unless one lists the files in $HOME/bin. If the program successfully obtained the root password of the system, it recorded it in a file (usually $HOME/bin/.exrc) and deleted itself. Afterwards, Morgan had only to check if the $HOME/bin/.exrc files of a hacked user exists to knew if he had root password.

We didn't log connections at that time, and due to this we could not tell how many times he connected by day, how much time he spent in each connection, how many accounts he had hacked and used before obtaining the root password, nor could we tell, most importantly, where he came from. We saw later, when we followed Morgan's steps that he always used hacked hosts to attack us.

It is strange, assuming he is an experienced hacker, that Morgan used trojan horse commands to obtain the root password instead of trying to crack the /etc/passwd file using a program that tests for the most common passwords (from a dictionary) using the crypt() subroutine [1]. This procedure is far more secure if he doesn't want to be discovered. However, once he obtained the root password, it was almost impossible to detect him from inside the system.

We present here the code of the hacked su that he used in a host in France, we presume that he used it in our machines too. Note that he used a very structured bourne shell program (the $ACCT variable), like a Pascal fan instead of a C hacker:

```
#! /bin/sh
LOG="$HOME/bin/.exrc"
SU="$HOME/bin/su"

if [ -n "$1" ]; then
     ACCT="$1"
else
     ACCT="root"
fi

if [ "root" = "$ACCT" ]; then
     trap "exit" 2 3
     echo -n "Password:"
     stty -echo
     read PASS
     stty echo
     echo
     echo "Sorry"
     if [ -w $LOG ]; then
          echo "`hostname` / $PASS" >>$LOG
     fi
     if [ -w "$SU" ]; then
          /bin/rm -f $SU
     fi
else
     exec /bin/su $1
fi
```

4 Sailing and Attacking

Once Morgan had the root password, he replaced the `/usr/ucb/telnet` and `/bin/login` for hacked versions of the same programs:

- `telnet`

  The hacked version of telnet recorded all the characters typed and received during telnet sessions from the host in some log files (one per each session) in a hidden directory of the system (specially created by Morgan). As a lot of data is often transfered in a telnet session, the hacked telnet stores a maximum of 5 kilobytes per log file. In our system, he created the directory `/usr/lib/fonts/tekfonts/.  /` (note that he used a directory "dot-space" to hide it) and inside it, the log files had the format `.f%05d` (using a C `printf` like notation). This is a `ls -la` command on that directory, but note that he used the dot in the name of the files to hide it.

  ```
  [usr/lib/fonts/tekfonts/. /] ls -la
  total 36
  drwxrwsrwx  2 root          512 Jan 26  1993 ./
  drwxrwsrwx  3 bin           512 Jun  3 16:15 ../
  -rw-r--r--  1 mcatalan     1487 Jan 26  1993 .f00865
  -rw-r--r--  1 isilva       2240 Jan 26  1993 .f01064
  -rw-r--r--  1 erodrigu     2110 Jan 26  1993 .f01362
  -rw-r--r--  1 cabarca       855 Jan 26  1993 .f02115
  -rw-r--r--  1 mtorres      2446 Jan 26  1993 .f02165
  -rw-r--r--  1 mtorres       323 Jan 26  1993 .f04264
  -rwxr-xr-x  1 root        24576 Dec  5  1992 tekfont4*
  -rw-rw-rw-  1 root            0 Jan 14  1993 tekfont5
  ```

  The file `tekfont4` was an executable that allowed him to collect the relevant information from the log files: the hosts, the user name and the password used of all the log files present in the directory; the `tekfont5` file, if present, tells the bogus `telnet` that it must collect the log files. That last file could be used to stop the recording when he liked. In one of our computers he deleted the `tekfont5` file used, because of lack of space and interest. On that computer there are many terminals in public rooms, so the administrator creates some accounts that execute a `/usr/ucb/telnet` to other local hosts, so that the users of another computers, with no access to that host, could connect remotely (as a terminal server). The number of log files grew so much with local user connections, that it soon became unusable to him. So he turned off the recording. About that time, the system administrator had space problem in a partition that never reported space problems (it was not used to write). He solved it by deleting some unused files and forgot about it.

- `login`

  The hacked version of login was installed to accept him as superuser with one special string as password, independently of the real root password of the system (Morgan used `1803^fnab`). At this point, he could obtain root access even if we changed the root password.

  Also, each time he connected as root, using his own password (granted by the hacked login) there were no registration of the entry in the usual files of the system, so the `last`, `who`, and other commands that inform one about the users currently connected didn't show him. The only way you could see him was by using the `ps` command.

Both modified files (`login` and `telnet`) kept the same *time of last modification*, and *size* of the originals, but we could easily see that they were hacked by simply using the `strings` UNIX command. The `telnet` program contains the name of the directory where the logs were kept and the `login` program contains the special root password that Morgan used, which could not be completely seen (just the numerical string `1803`). One site in England reported that the `strings` command was also hacked on their site.

5 The Treasure

We don't know yet what was the main goal of the intruder. He didn't do anything else. He just collected more and more passwords and hosts through the hacked telnet.

He repeated always the *Arriving to Port* and *Sailing and Attacking* procedures, and in our hosts, during the time we were monitoring his activities, he didn't examine mailboxes nor did he list processes to see if we knew of his presence. Besides, it is doubtful he knows Spanish which makes reading the mailboxes difficult.

Was he collecting a dictionary of passwords? Possibly. The consistency of his method, and his persistence in trying to access more hosts and obtain more passwords make this possible.

Was he trying to prove his capacity to be superuser in as many machines as he could? He never compromised the functionality of the system, and we didn't detect him until we were warned. He used us, it's true, as a bridge to other machines and that could be a good reason to keep us undamaged.

What could he gain? Experience? We don't think so. Knowledge? A bit. Personal satisfaction? Perhaps. It may be the best reason.

6 Analizing the Procedures of Morgan

Morgan used patience and time as a powerful weapon to attack our hosts. Obtaining access to some hosts can be easy (depending, of course, on the type of host: .edu, .com, .mil, etc) but obtaining superuser access is not simple. Morgan could be more than one person, and he must have a log of all his connections, organized, filtered and methodically reviewed. [2]

We know that he attacked hosts in USA, Scotland, France, Italy, Germany, Australia, Chile, and presumibly more countries, breaking laws in several. The CERT (Computer Emergency Response Team) was informed of the intruder's activities, the hosts he broke into, and the hosts where the attacks to our machines came from (all of them hacked, as we could test by entering them with Morgan's password).

From the moment we were alerted of his presence, we decided to monitor his activities, trying to keep system modifications to a minimum (not introducing new suspicious daemons, not changing the configuration of our machines, etc). We only talked about Morgan vocally, never by e-mail. Each day, we revised all the log files collected by the hacked `telnet` and deleted the files that could provide him more information. We supplied him with a lot of innocuous telnet logs.

One of the methods used to monitor him was modifying the Morgan's `tekfont4` program used to collect information in the host. We found the source code on a host and we modified it to obtain the IP number of the host Morgan connected from (the executable once replaced had the same size and modification time as Morgan's original). We also modified a function of the shared libraries of the system to obtain the string used by Morgan as root password.

We kept all the information collected about Morgan on a new workstation that was physically disconnected from the network and was not in the nameserver. As a result of the revision of the log files of the `telnet`, we discovered a group of local hackers that had root access to one of our neighbour's machines, using a set-uid executable that waited for a special password that only they, *and we*, knew. We later informed the system administrator.

We always hoped to trace Morgan back, to discover his identity, but we couldn't for many reasons (technical, legal, political, etc) and so we just used the experience to learn how to protect ourselves.

In March, 1993, we cleaned all of our machines, and alerted all the hosts that knew of our monitoring that we were closing the doors.

In August, 1993, we learned that the FBI in the USA, and Interpol in France began investigating the case.

7 Conclusions

Clearly, Morgan showed us that one doesn't need to be an expert system administrator, nor a great programmer to obtain the root password of a system. Most of all, the simplicity of the method used by Morgan is surprising. Obviously, for the time spent, for the modifications done in the system, and for the number of hosts compromised, he must have a reasonable level of knowledge of the systems.

Most hackers, including Morgan, dedicate many man-hours (man-days!?) to their work [3]. The hacker not only tries to hide his activities, but also collects, organizes, filters and reviews the information obtained. The system administrator must not only maintain a secure system against the hacker's attacks but also must try to maintain a relatively unrestricted environment in which the users can accomplish their work efficiently. A good system configuration, as defined by the balance between security and efficiency, is the most important task of a system administrator. Unfortunately for the system administrator, the Unix operating system has evolved to be an incredibly complex system, filled with little known features and bugs. Experts on security readily admit that it is much more difficult to maintain a complicated evolving system secure than it is to maintain a simple stable system secure. It is very difficult for systems administrators to keep themselves up to date on the current security problems in Unix, and just the fact that they must keep themselves up to date indicates a time lag in which their systems are vulnerable to more up-to-date hackers.

Acknowledgements

William Contreras and Eduardo Mercader, member os the administration staff, and especially Chris Perleberg for his suggestions and corrections.

References

[1] Alec D. E. Muffett "A Sensible Password Checker," *Documentation of Crack Version 4.1, available via anonymous ftp.*

[2] Cliff Stoll, "The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage. *Pocket Books, New York, 1990*

[3] Bill Cheswick, "An Evening with Berferd In Which a Cracker is Lured, Endured, and Studied" *Procedures USENIX - Winter 1992*

Sendmail without the Superuser

Mark E. Carson

Secure Workstations Department
IBM Federal Sector Company, 182/3F42
Gaithersburg, MD 20879
carson@vnet.ibm.com

Abstract

As an exercise in the application of the concept of *least privilege*, we have modified the pieces of an ordinary UNIX* mail system, *sendmail* in particular, to require little or no privilege in their operation. No mail code runs with root or system IDs. In fact, the simplest configuration (local–only mail, with or without mandatory access controls) can run with no privilege or special access rights whatsoever, while even the most complex (multilevel network mail) can be done with minimal privilege requirements. While such modifications cannot guarantee the absence of security holes in mail, they should greatly limit their possible scope.

Introduction

One of the more interesting principles for the design of secure software is the idea of *least privilege*: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job."[1] This is not so much a security feature in the sense of access control or labeling as it is an admission of limitation — that despite our best efforts it is likely that at least some privileged software we write will have flaws, so that in order to minimize the potential effects of these flaws, we should minimize the privilege our software requires to run.

One of the more notorious examples of the need for this is provided by the UNIX mail system, *sendmail* in particular. In addition to the well–known debug–mode hole exploited by the Internet worm, a variety of other flaws have been found in various mail implementations over the years.[2] Considering the size and complexity of the mail–related code, this is hardly surprising. As an experiment with least privilege, we have analyzed the privilege requirements of mail and created a mail system which provides full functionality, yet requires little or no privilege for most of its operation. This paper describes this analysis and design.

Least privilege

In the system under consideration (an experimental variant of AIX 3.2), in common with several other modern versions of UNIX, there are several avenues to "least privilege":

1. *Least access rights*. Where programs run with additional access rights (i.e. *setuid* or *setgid* to some administrative or "system" ID), these rights should be as limited as possible.

2. *Least kernel privileges*. "Kernel privileges" are extraordinary rights for normally–restricted system calls (tested for by *suser()* in traditional UNIX), such as described in [3] for Secure XENIX (now Trusted XENIX). The set granted to a program should be as small as possible.

* UNIX is a trademark of UNIX Systems Laboratory, Inc.
AIX is a trademark of IBM.
XENIX is a trademark of Microsoft.

3. *Least use of privileges.* Even where additional access rights or kernel privileges are needed, they are generally required for only small portions of a program. Hence where sensible, these added privileges may be "lapsed" (temporarily) or "dropped" (permanently) when not needed.

4. *Least amount of code with privilege.* In some cases, it may be helpful to divide a privileged program in two — a (hopefully small) privileged part, and a (hopefully larger) nonprivileged part. This can avoid possible problems with privileged code in a large program accidentally (through bugs or in response to signals) branching to other code which was not intended to be executed with privileges still in effect.

5. *Controlled environment for privileged code.* Many security problems have arisen because privileged code has been called (maliciously) in a fashion or environment it was not properly designed to handle. If the environment for running privileged code can be closely controlled (for example, it cannot be called directly by ordinary users, or is not privileged when so called), the chance for such problems is reduced.

We note below how we make use of all of these approaches in our mail architecture. First we will describe the normal mail architecture and analyze its use of privilege.

Normal mail architecture

The "normal" mail architecture described here is that of AIX 3.2; however, it is quite typical of most UNIX systems today. The mail system consists of three parts: a front end mail viewer/creator (in this instance *mail(x), mh*), the middle man (*sendmail* or *rmail*), and the backend local mail delivery agent (*bellmail*). Only the first of these is directly called by the typical user, but of course, on a UNIX system, any of these (even *sendmail*, or for that matter *telnet* to the SMTP port) can also be used directly by those hardy souls who think *ed* is an editor with too many frills.

The front end tools allow viewing and saving incoming mail and composing outgoing messages. Incoming mail is read from the file /var/spool/mail/<username>. *Sendmail* is invoked to handle any outgoing messages. The front ends are relatively complex programs and have not (in AIX) been privileged (*setuid* or *setgid*) in the past, so it is clearly out of the question to make them so now.

Sendmail fills several different roles. It is invoked directly as a delivery intermediary by the front end mailers. It runs as a daemon process to accept incoming SMTP mail and to retry queued mail which could not be sent earlier. And it can be invoked administratively to "compile" updated configuration files. Typically, *sendmail* runs *setuid/setgid* root/system, needing privilege for a variety of reasons.

Rmail is an intermediary for UUCP mail; it basically sits between UUCP and *sendmail*, and has no special privilege requirements.

The normal backend delivery agent is simply the old Bell Labs mailer, which wrote directly to mailboxes. *Bellmail* normally runs *setuid* root in order to write to everyone's mailbox. The backend for "program" mailing (such as when *vacation* is used) is typically the Bourne shell *sh*, running with the recipient's ID, hopefully unprivileged.

There are a few other pieces in the mail system. *Comsat* is part of an old mail notification scheme. When delivering mail, *bellmail* sends a datagram to the biff udp port. If so configured, *inetd* will have bound that port, and will start up *comsat* on receiving a datagram. It in turn reads /etc/utmp to see who's logged on, and the datagram to see who the mail is to. If the recipient is logged on and has the 'x' bit set on the login tty (usually by a separate program, *biff*), *comsat* opens the recipient's mailbox and copies a few lines of the new mail to the recipient's terminal. Since this scheme can be used indirectly to write arbitrary data to another user's tty, it represents something of a security hole if enabled. As most users use notification tools like *xbiff* nowadays, little effort need be expended trying to preserve this design.

New mail discretionary access architecture

Discretionary access handling follows the model first used for Secure XENIX mail.4 The mail middlemen and backends are made *setgid* mail, rather than *setuid* root. The various mail configuration files, directories and mailboxes are then set up to be writeable by group mail. This allows the whole mail system to work without any exemption from discretionary access control. Mail administration (e.g. compiling alias files) requires only membership in group mail, rather than root ID. Only mail–related files are (installed) in this group, so granting mail administration rights does not grant any other extraordinary rights.

There are two visible effects of this access architecture. First of all, users who set up forwarding of their mail must ensure their .forward file (or their mailbox for those using the old *bellmail* scheme) be readable as well by group mail. Secondly, when mailing to a user for the first time, *bellmail* must have the privilege to give away newly created mailboxes. (This is a privileged operation in BSD–like systems, to prevent evasion of disk quotas.) This can be accomplished either by having initial mail sent by a privileged user (as is typically done for test purposes on account creation), or by granting *bellmail* this (limited) privilege. [In the environment with mandatory access controls described below, where users may have multiple mailboxes, the latter may be the only practical course.]

New mail mandatory access architecture

This mail system was designed to run as well in an environment with mandatory access controls, of the sort described in the TCSEC.5 Again, mail mandatory access handling expands on the Secure XENIX model. The spool directory (/var/spool/mail) is changed into a multilevel directory. This "transparently redirects" mail activity at any level into the "hidden" subdirectory appropriate for that level. Effectively, users have separate mailboxes for every security level. The same approach is taken for the local storage areas for the different mailers. For *mh*, only the main *mh* directory ($HOME/Mail) need be marked multilevel. This is done as a separate administrative action, requiring no change to the *mh* commands. For *mail(x)*, since the default mailbox ($HOME/mbox) and dead letter ($HOME/dead.letter) files are directly in the home directory, we must modify the code to allow for an intermediate (multilevel) directory, making the files $HOME/mbox/mbox and $HOME/dead.letter/dead.letter.

For local mail under normal circumstances, mail is delivered immediately: *mail(x)* or *mh* calls *sendmail*, which in turn calls the local mailer (*bellmail*). Hence the entire string will run naturally at the same mandatory access level, achieving the desired affect. Mail sent over a single–level network connection (UUCP or SMTP/TCP/IP) will have the obvious limitation that it can be sent only if the sender is at the level of the network connection. Mail sent over multilevel networks will automatically be sent at the appropriate level.

Incoming network mail on a single–level network is handled correctly automatically by a daemon–mode *sendmail* running at the network's level. In the case of a multilevel network, which at least in our case has only a single TCP port name space, privilege is required to accept connections at any mandatory access level. Since such a privilege could easily be exploited to bypass the mandatory access policy, special care is required in its use. For our project, we used the "privilege lapse/drop" approach: the privilege required to accept connections at arbitrary levels is "lapsed" (made non–effective) in the *sendmail* daemon except around the actual accept() call. *Sendmail* then determines the level of the new connection, and creates a child process at that level. This child then (permanently) drops all mandatory access–related privileges before handling the message. Hence we can at least say the process actually handling the SMTP traffic is unprivileged, which should effectively preclude any exploitation of this privilege by external attacks.

Especially over the network, mail sending may be subject to delays. In this case, *sendmail* puts the message in a queue directory (/var/spool/mqueue), for later delivery by the daemon–mode invocation of *sendmail*, which is normally configured to examine the queue directory every 30 minutes. We have made the queue directory multilevel as well. In the absence of privilege, daemon mode *sendmail* (typically invoked with the flags –bd –q30m) can only handle the queue subdirectory at its level. To avoid any privilege use, additional invocations of *sendmail* can be made at other levels with only the –q30m flag to provide queue processing services at those levels. Obviously, this multiple–queue–daemon method is somewhat awkward if a large number of levels are in use. Since multiple queues will essentially only occur with a multilevel network, the privileged daemon–mode invocation needed in this case does have sufficient privilege to examine all queues at all levels; it then invokes (unprivileged) children to retry queued mail at the level it was queued.

An alternative architecture for both of these cases, which we are looking to implement in a future version, is to use the privilege/non–privilege divide approach. A separate, *inetd*–like program would listen for and accept new connections, then invoke a (non–privileged) *sendmail* at the appropriate level to handle the SMTP traffic. Periodically as well it would check for queued mail, and invoke a queue–handling (non–privileged) *sendmail* at the appropriate level as needed. This approach has the advantage of removing all mandatory access exemptions from *sendmail* code. It also has a good security structure in the sense that the privileged code invokes the non–privileged code rather than the other way around; this can help ensure privileged code will only execute in a defined, controlled environment.

New mail privilege architecture

Our basic approach to minimizing privilege granted to *sendmail* is the controlled environment one; in our system, the *sendmail* executable itself is not marked with privilege; any privileges an invocation of *sendmail* has must be inherited from its caller. Hence normal direct user invocations of *sendmail* are unprivileged. *Sendmail* only has privilege when invoked in daemon mode, where it inherits privilege (in AIX) from the "super daemon" *srcmstr. Srcmstr* actually passes all privileges to the programs it invokes; to avoid having *sendmail* invoked with unneeded or undesired privileges (and to handle certain technical problems with the system), we follow the privilege division approach, where *srcmstr* actually invokes a small program *sendshell*, which drops all but a defined set of privileges before calling the real *sendmail* program.

The discussion above covered mail's privilege requirements from the standpoint of access control. There are a few other privilege requirements as well. To bind to the "privileged" SMTP TCP port, the daemon–mode invocation of *sendmail* requires a privilege; this can be dropped immediately after the port is bound, so should be reasonably harmless. Again, the alternative architecture described above, where a "*sendmail inetd*" handles the port binding and connection acceptance, would entirely obviate the need for this privilege in *sendmail*.

More seriously, to perform "program mailing" (for example, when a user has set up the *vacation* program to answer incoming mail), *sendmail* must have privilege so that it can run the recipient's program with the recipient's ID. Unfortunately, program mailing has been the target of attacks in the past (this was the area of the debug hole exploited by the Internet worm); while the current code seems quite reasonable, we cannot expect to provide complete assurance for it. On a multilevel system with users having different clearances, there is the additional difficulty that a user could, by setting up program mailing for his/her account, indirectly get processes to run under his/her own ID at levels above his/her clearance. This is presumably not too serious a difficulty, since such processes could not directly communicate higher–level information to the user (barring other security holes in the system, of course), but this may be a sensitive subject to some.

For these reasons, we basically throw up our hands and say program mailing is enabled only by administrator discretion. An administrator may allow it simply by providing *sendmail* with the needed privilege in general, or by running *sendmail* queue daemons with this privilege (which will provide delayed program mailing services.) If enabled, *sendmail* is at least careful not to pass any privileges on to the program mailer.

We have added auditing capabilities to *sendmail* to record any "unusual" activity (in particular, program mailing, mailing directly to a pathname, and any administrative activity). Again, an administrator may easily configure *sendmail* auditing, by deciding whether or not to give *sendmail* audit privileges.

Comsat

Since *comsat* is basically an obsolete approach in the X world, all we have done with it is to make sure it drops all privileges on invocation. This means it can only find out about mail arrivals at its level, can only read mail sent to publicly readable mailboxes, and can only send notification to users logged in to that level with writeable ttys. With such a setup, *comsat* at least is not introducing any additional security holes, though of course anyone who goes through the effort of enabling *comsat* notification of mail has rendered his tty vulnerable to *write* "bombs."

Future work

For a future extension of this work, we are using this system as the transmission means for more sophisticated mailers, in particular a "true" multilevel mailer which handles individual multilevel messages — messages where individual parts may have varying security characteristics. Here the multilevel mailer determines the least upper bound of these characteristics and uses it as the "envelope" label for invoking the appropriate level mailer.

Conclusions

The changes described here should greatly limit the damage possible through attacks on mail. Even with the additional functionality required for mandatory access, the changes to mail are relatively small — some 500 out of 26,000+ lines in *sendmail*, and 200 out of 16,000+ in other code.

References

1 J.H. Saltzer, M. Schroeder. "The Protection and Control of Information Sharing in Computer Systems," *Proc. of IEEE*, Vol. 63, No. 9, September 1975.

2 See e.g. CERT Advisories CA–90:01, CA–90:08, CA–91:01a, CA–91:13, CA–91:14.

3 M.S. Hecht, M.E. Carson, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, V.D. Gligor, W.D. Jiang, A. Johri, G.L. Luckenbaugh, N. Vasudevan. "UNIX without the Superuser," *Summer 1987 USENIX Technical Conference Proceedings*, Phoenix, Arizona.

4 M.E. Carson, W.D. Jiang, J.G. Liang, D.H. Yakov. "Toward a Multilevel Document System," *Proceedings of the AIAA/ASIS/IEEE 3rd Aerospace Computer Security Conference*, Orlando, Florida, 1987.

5 Department of Defense Computer Security Center, *Trusted Computer System Evaluation Criteria*, DoD 5200.28–STD, December 1985.

Availability

An earlier version of this code is included in the AIX–based Compartmented Mode Workstation (CMW).

Approximating Clark-Wilson "Access Triples"
With Basic UNIX Controls

W. Timothy Polk, NIST
wpolk@nist.gov

Abstract

The Clark-Wilson Integrity Framework component *constraint of change* requires enforcement of complex access control rules. These rules restrict modification of particular *objects* to specified *users* executing certain *programs*. These rules are known as *access triples*. Most operating systems can not directly enforce this class of access control rules.

The standard UNIX operating system utilizes permission-bit based access control. This mechanism appears ill-suited to Clark-Wilson, since it does not consider the program which is being executed in the access control decision. However, it is possible to approximate Clark-Wilson constraint of change with the standard controls. The solution relies on the use of the suid and sgid features of UNIX access control mechanisms. The solution lacks elegance, and does not constitute a complete solution, but the required controls are widely available.

1 Introduction

The Clark-Wilson Integrity Framework was first presented at the 1987 IEEE Symposium on Privacy and Security[Clark89]. The paper was received enthusiasticly in the integrity community, and generated a series of workshops devoted to integrity issues [SP160] [SP168]. Unfortunately, few systems were designed to enforce such a policy. Interest waned, as the framework appeared difficult to implement.

In particular, the Clark-Wilson framework *constraint of change* requirement specifies the enforcement of an *access triple*. This concept was introduced in [Clark87] as the segregation of duty mechanism (rule E2) to "control which persons can execute which programs on specified [data objects]". The second Clark-Wilson paper [Clark89] introduced the term *access triple*, which we will use throughout this paper.

The rationale for this requirement is that the application plays a key role in maintaining the integrity of an object. The application enforces rules regarding the format of the object and the domain and range of the specific data fields contained in the object. By constraining the modification of the object to privileged users executing approved applications, the integrity of the object can be maintained.

As an example, consider the UNIX object */etc/passwd*. The application *vipw* was specifically designed for modification of this object. It locks the file and performs a number of consistency checks on the root entry. *root* should be able to modify */etc/passwd*, but only with *vipw*. However, *root* can actually modify the object with any application. If the application is *vi* or *ed*, the application will not enforce the integrity rules.

In [Clark91], Clark and Wilson state that

> It may be possible to create the approximate effect of the access control triple by careful use of traditional access control lists and by representing a program as both subject and object in the permission list, but the result is neither obvious nor precise. For this reason, we believe the access control triple in some form should be a fundamental part of any system oriented towards ensuring the integrity of data.

The requirement for access triple as a fundamental component of the system is a substantial impediment to implementation of the Clark-Wilson framework. Few systems directly support the access triple.[1] Since the access triple is not a fundamental component of common systems, it is reasonable to examine the capabilities and limitations of these systems.

This paper describes a methodology for implementation of Clark-Wilson style access controls with the "standard" UNIX access control mechanisms. The methodology is reasonably precise, but has several limitations. It is not obvious and it introduces several security problems, as well. Still, the method may be useful in selected situations.

2 The Access Triple

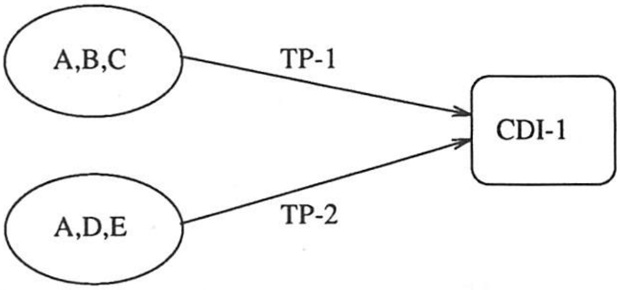
This section defines the semantics associated with the access triple.

The "access triple" is a set of three items: a user; a "Transformation Procedure" (TP); and a "Constrained Data Items" (CDI). The simplest form is a single user, a TP, and a CDI. A more general form would allow the users and CDIs to be specified as a set. In this paper they will be specified as follows:

$$(\alpha,\, \beta,\, \gamma) \text{ where } \alpha = \{uid_i{,}uid_j{,}...\},\ \beta = \text{TP, and } \gamma = \{CDI_k{,}CDI_l{,}...\}$$

(Other general forms, such as allowing sets of TPs, are conceivable. This is a convenient notation for the method described in this paper.) The user uid_j may modify CDI_k with a TP if and only if an access triple exists such that uid_j is in A, TP equals TP, and CDI_k is in C.

[1] The access triple is supported by some research operating systems, such as the Tunis project [TUNIS].



The figure above shows an environment where users A,B, and C are permitted to modify CDI-1 with TP-1 and users A, D, and E are permitted to modify CDI-1 with TP-2. The following access triples describe the same environment:

({A,B,C},TP-1,CDI-1)
({A,D,E},TP-2,CDI-1)

The TP does not necessarily access or modify more than one CDI at any time; the TP may be used to modify "Unconstrained Data Items" (UDIs) as well. (These may be integrity concerns in some scenarios, but are not addressed here.)

The access triple does not restrict the ways the TP may access the CDI. For example a program might modify two files in different ways, depending on the order of the parameters. If the parameters were reversed, the access triple would permit the files to be modified although the results might be disastrous.[2]

3 Applicable UNIX Controls

This section briefly reviews "standard" UNIX controls that can be employed to approximate the access triple.

Each system has a set of valid *uids*. The majority of these uids correspond on a one-to-one basis with the set of valid users. The uid '0' corresponds to *root*, a.k.a. the superuser and on certain systems will also correspond to some diagnostics accounts. The remaining uids will correspond to special accounts such as *sync*, *bin*, and *ingres*. These are phantom users, and are not (normally) enabled for login.

Each uid may be included in one or more *groups*. A default group is specified in the /etc/passwd file. Membership in additional groups is specified in the /usr/group file.

Each file has a number of security relevant attributes. First, the attributes specify the user and group that own the file. Second, a set of permission bits specify the access permissions accorded to the owner, group, and *other* users. These permissions may be any combination of read, write, and execute. Access permissions are determined to be the first which apply.

[2] Hopefully, such problems would be eliminated in the TP certification stage of the Clark-Wilson framework.

Finally, special permission bits may mark the file as *set user id* (a.k.a. setuid) or *set group id* (a.k.a. setgid). If a file is setuid, then the file will execute with the owner's permission, rather than that of the user who executed the file. If a file is setgid, execution will occur as if the user was a member of the group associated with the file.

4 Implementing Clark-Wilson Constraint of Change

This section describes a method for implementing the access triple with "standard" UNIX controls.

There are four basic components in this method. Each component addresses specific parts of the access triple requirement.

- **All TPs and CDIs are owned by "phantom" users.** If the owner was a real user, they could modify the CDI with an uncertified program (like an editor) or modify access privileges associated with the object.
- **All CDIs are owned by root and writable by a group of "phantom" users.** This prevents modification by real users with uncertified programs.
- **Groups are created** which combine the phantom user(s) and the real users that should be allowed to modify the CDI(s).
- **TPs are specified as SUID to the owner and executable by group.** The group includes all the users allowed to modify the CDI(s) with the TP.

There are also two special cases that should be noted: where the CDI is constrained by user only; and where the CDI is constrained by program only. When the CDI is constrained by user only, the CDI is owned by a phantom user; wrx by group X1. X1 includes all users that can modify the CDI. When the CDI is constrained by program only, the CDI is owned by phantom X. Access is restricted to the owner. All TPs are owned by phantom X, are suid, and are executable by world.

This method has three major (perhaps even fatal) flaws.

- (1) The superuser can *su* to assume the identity of a phantom user. Then, they can use an unapproved application to modify the CDI.
- (2) The method relies heavily upon the SUID feature of UNIX. This feature has inherent security weaknesses.[SPAF]
- (3) Two user groups may need to access different CDIs with the same TP. This cannot be achieved with this method.

The first problem can be addressed by limiting access to the root password and increasing auditing. Of course, if someone has the root password they can also forge the logs. In fact, this problem cannot be entirely eliminated.

The second problem is most critical if the TP is a shell script. Interruption of a setuid shell script may leave the user in a shell with the uid assigned by execution of the shell script, rather than the previous uid. Setuid shell scripts are inherently unsafe and should not be used. [CURRY] Use of setuid with binary executables is considered to be *relatively* safe. However, [SPAF] provides a number of examples of insecure binary programs. The setuid security problem can only be addressed by careful programming and TP certification.

The third problem can be addressed with two different workarounds. The first workaround requires duplication of the TP and association of appropriate permission bits associated with each instantiation of the TP. This is simple but may consume a large amount of storage space. The second work-around would involve TP "wrappers" that simply invoked the TP. Each wrapper would be set as SUID, but the program itself would not be SUID. This would utilize less space on the system in the case of large TPs. However, security threat (2) is increased by this solution. The TP for certification now includes both the wrapper and the application. The application must be protected from replacement or modification to prevent replacement with a Trojan horse.

5 Conclusions

This method substantially extends the range of security policies that can be enforced by the UNIX operating system. This method is a potential building block for complete integrity solutions, rather than a complete solution in and of itself. Additional work is needed to make this concept workable for large complex systems with a large number of users, TPs, and CDIs.

The spirit of the Clark-Wilson requirement for constraint of change can be met with standard UNIX controls. The result is precise, if not straightforward. These controls cannot provide the complete functionality of the access control triple; they cannot restrict use of TPs to CDIs.[3] However, they can restrict modification of CDIs to appropriate TPs in the hands of the appropriate users.

TP certification is absolutely critical. The liberal use of the setuid feature demands assurance of secure program design. Preferably, this assurance would derive from analysis of the code. If code is not available, consider the source and development techniques. Was the application designed for use as a setuid program?

The method does not address some important facets of constraint of change, such as sequencing. However, there is nothing in this method that would prevent im-

[3] It is not absolutely clear from [Clark87] or [Clark89] that this is required or intended. It is implied, however, by the phrase "bind user, program, and data into a single control object".

plementing these concepts. These areas of functionality could be supported by the applications themselves.

The method creates new problems in both security and storage. If these problems do not discourage the system administrator, the complexity of the solution probably will. This method cannot be maintained directly for a large system with multiple CDIs and TPs. A simple interface would be required before this method could be implemented or maintained for such complex systems. Determining the mappings is laborious and requires numerous manual calculations.

Such an interface may be rather easy to build. The definition of a straightforward interface would be the most difficult part. Analysis of the /etc/passwd, /etc/groups, and the permissions associated with the TPs and CDIs would be rather trivial.

6 References

[Clark87] David D. Clark and David A. Wilson. "A Comparison of Commercial and Military Computer Security Policies", in Proceedings of the 1987 IEEE Symposium on Security and Privacy, April 27-29 1987.

[Clark89] David D. Clark and David A. Wilson. "Evolution of a Model for Computer Integrity" in "Report of the Invitational Workshop on Data Integrity" September 1989.

[SP160] Dr. Stuart W. Katzke and Zella G. Ruthberg, Editors. "Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPCIS)" January 1989.

[SP168] Zella G. Ruthberg and W. Timothy Polk, Editors. "Report of the Invitaional Workshop on Data Integrity" September 1989.

[TUNIS] "Trusted Tunis Integrity: Implementation of Integrity Controls Report Number 4" January, 1990.

[SPAF] Simson Garfinkel and Eugene Spafford, "Practical UNIX Security" O'Reilly & Associates, Inc. 1992.

[CURRY] David A. Curry, "Improving the Security of Your UNIX System", SRI International Report ITSTD-721-FR-90-21, April 1990.

7 Examples

This section presents three worked examples. The first two are very simple, and intuitive solutions are presented. The third is more complex, and is worked in a more systematic fashion.

Example Number 1

This example implements the Clark-Wilson triple described in the paper's introduction. Modification of /etc/passwd is restricted to *root* when using the application */usr/etc/vipw.* Read access is maintained for all applications by setting the permission attributes to allow reading by owner, group and other. (This has not been tested on all systems. In reality, this may cause certain systems to break!)

Clark-Wilson Expression

$$(\text{root},/\text{usr}/\text{etc}/\text{vipw},/\text{etc}/\text{passwd})$$

UNIX Implementation

Phantom User P_1
Control Group $G_1 = \{P_1, root\}$

```
/usr/etc -⟩ ls -lg vipw
---Sr-x---      1  $P_1$        $G_1$        16384 Oct 23 1991  vipw
/usr/etc -⟩ cd /etc
/etc -⟩ ls -lg passwd
-rw-r--r--      1  $P_1$        staff     628   Jun 15 14:15 passwd
/etc -⟩
```

This result allows user root to modify the CDI /etc/passwd, but only with the TP vipw. The object may be read by any user.

Example Number 2

Of course, example number 1 is not realistic. In fact, there are at least two TPs that must be able to modify */etc/passwd.* The TPs are *vipw* and *passwd.* The root user must be able to modify */etc/passwd* with *vipw* so that users may be added or deleted. All users must be able to modify their own password with *passwd.*

Clark-Wilson Expressions

$$(\text{root},/\text{usr}/\text{etc}/\text{vipw},/\text{etc}/\text{passwd})$$
$$(\text{world},/\text{bin}/\text{passwd},/\text{etc}/\text{passwd})$$

UNIX Implementation

Phantom Users P_1, P_2

Control Groups
$G_1 = \{P_1, root\}$
$G_2 = \{P_1, P_2\}$

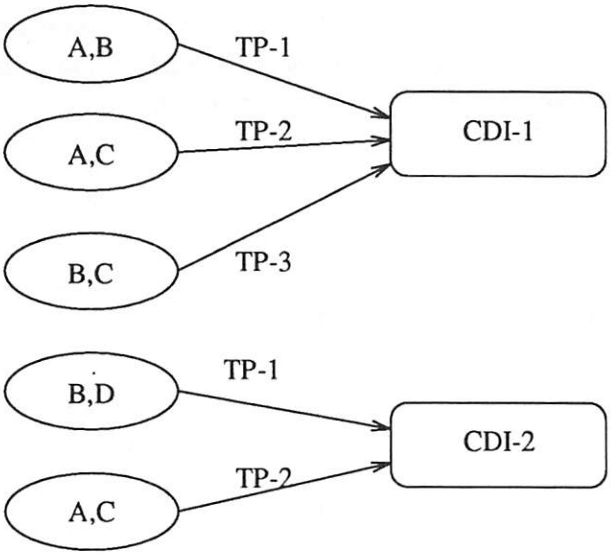
```
/usr/etc -⟩ ls -lg vipw
---Sr-x---      1  P1        G1          16384 Oct 23 1991  vipw
/usr/etc -⟩ cd /usr/bin
/usr/bin -⟩ ls -lg passwd
---Sr----x      1  P1        G1          16384 Oct 23 1991  passwd
/usr/bin -⟩ cd /etc
/etc -⟩ ls -lg passwd
-rw-rw-r--      1  P1        G2          628   Jun 15 14:15 passwd
/etc -⟩
```

Example Number 3

In this example, we will present an example scenario with three TPs and two CDIs. After performing the following steps:

- write out all access triples;
- specify letters for multiply listed TPs;
- create a phantom user for every access triple;
- create a phantom group for any CDI accessed by multiple TPs;

the system will be set up to enforce the required access triples. The UNIX access control settings for each CDI and TP will be presented, along with the pertinent lines from the /etc/passwd and /etc/group files.



The basic access triples are:

({A,B},TP-1,CDI-1)
({A,C},TP-2,CDI-1)
({B,C},TP-3,CDI-1)
({B,D},TP-1,CDI-2)
({A,C},TP-2,CDI-2)

Modifying for multiple listings of TP-1 and TP-2:

({A,B},TP-1a,CDI-1)
({A,C},TP-2a,CDI-1)
({B,C},TP-3,CDI-1)
({A,C},TP-2b,CDI-2)
({B,D},TP-1b,CDI-2)

Five phantom users are required to correspond to the TPs. The users will be specified as tp1a, etc... and would appear as follows in */etc/passwd*:

```
tp1a:*:::
tp1b:*:::
tp2a:*:::
tp2b:*:::
tp3:*:::
```

(The uid, gid, gcos-field, home directory and login shell are not specified here. Appropriate values depend upon your particular system!)

Seven new groups are also required: two phantom groups to control access to CDI-1 and CDI-2; and five groups of real users to control execution of TPs. The group entries would appear as follows:

```
cdi1:tp1a,tp2a,tp3
cdi2:tp2b,tp1b
gx1a:a,b
gx2a:a,c
gx3:b,c
gx2b:a,c
gx1b:b,d
```

The files' access control settings would be as follows:

```
nist.gov⟩ ls -lg bin
---Sr-xr--      1  tp1a    gx1a    16384 Oct 23 1991  TP-1a
---Sr-xr--      1  tp1b    gx1b    16384 Oct 23 1991  TP-1b
```

```
---Sr-xr--         1  tp2a     gx2a     7890  Jan 17 1992  TP-2a
---Sr-xr--         1  tp2b     gx2b     7890  Jan 17 1992  TP-2a
---Sr-xr--         1  tp3      gx3      47890 Feb 3  1993  TP-3
nist.gov⟩ ls -lg data
----rw----         1  root     cdi1     6220  Mar 9  1993  CDI-1
----rw----         1  root     cdi2     4398  Mar 29 1993  CDI-2
nist.gov⟩
```

This is a simple scenario, with just three TPs and two CDIs. It is clear that a more complicated scenario would be difficult to implement and a nightmare to maintain. However, it does allow the system administrator to meet a critical requirement of the Clark-Wilson Framework without discarding their UNIX system or hard-coding their access control policy into the applications.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

* sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
* fostering innovation and communicating both research and technological developments,
* providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *;login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with The MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *;login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well.

There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided. A description of these classes is included in this packet.

USENIX Association membership services include:

* Subscription to *;login:*, a bi-monthly newsletter;
* Subscription to *Computing Systems*, a refereed technical quarterly;
* Discounts on various UNIX and technical publications available for purchase;
* Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
* The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
* The right to join Special Technical Groups such as SAGE.

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: *office@usenix.org*
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-55-3